آزمایش هفت

خلاصه ی آزمایش:

مروری بر شبکه و برخی دستورات مرتبط با آن

اهداف آزمایش:

- آشنایی با مفاهیم اولیه شبکه
- مروری بر TCP
- آشنایی با برخی دستورات در لینوکس مرتبط با شکبه
- آشنایی با socket programming
- مروری بر thread
- Programming with pcap

تجهیزات مورد نیاز:

- برد Raspberry pi II و کابل برق آن
- کارت Micro SD
- کامپیوتر (به همراه موس و کیبورد)
- کابل شبکه جهت ارتباط برد با کامپیوتر
- نرم افزار Putty جهت راه اندازی SSH

شرح آزمایش:

1- برنامه ای بنویسید که به صورت کلاینت به کامپیوتر متصل شده و روی یک پورت tcp داده ای را به کامپیوتر بفرستد.

2- با تغییر برنامه ی فوق و با استفاده از برنامه ی کپی که در جلسات قبل نوشته اید، برنامه ای بنویسید که یک فایل را به کامپیوتر منتقل کند.

3- برنامه ای بنویسید که به صورت کلاینت روی یک پورت udp داده ای را به کامپیوتر بفرستد.

4- برنامه ای بنویسید که به عنوان سرور باشد و دو کلاینت را که به آن متصل شده اند را echo کند. (اختیاری)

5- یک proxy server به زبان C در رسپبری پای بسازید و با browser کامپیوتر آن را چک کنید.

6- اینترفیس های مجازی با ip های مختلف برای رسپبری خود درست کنید و توسط کامپیوتر آنها را ping کنید.

7- برنامه ای به زبان سی بنویسید که کلاینت ها بتوانند به یک نرم افزار مستر متصل شده و پس از ثبت نام، بتوانند با یکدیگر چت کنند.

وظایف:

1- بخش مطالعه ی این دستور کار به طور کامل مطالعه شود.

مطالعه:

آشنایی با مفاهیم اولیه شبکه

مروری بر TCP

آشنایی با برخی دستورات در لینوکس مرتبط با شکبه

آشنایی با socket programming

مروری بر thread

Programming with pcap

آشنایی با مفاهیم اولیه شبکه

ه-۱- شبکه‌های کامپیوتری

یک شبکه شامل مجموعه ای از دستگاهها (کامپیوتر ، چاپگر و ...) بوده که با استفاده از یک روش ارتباطی (کابل ، امواج رادیوئی) و به منظور اشتراک منابع فیزیکی (چاپگر و ...) و اشتراک منابع اطلاعاتی به یکدیگر متصل می گردند. شبکه ها می توانند با یکدیگر نیز مرتبط شده و شامل زیر شبکه هائی باشند.

ه-۲- تفسیم‌بندی شبکه‌ها

شبکه های کامپیوتری را بر اساس مولفه های متفاوتی تقسیم بندی می نمایند. در ادامه به برخی از متداولترین تقسیم بندی های موجود اشاره می گردد.

تقسیم بندی بر اساس محیط انتقال

جهت انتقال داده در شبکه‌ها ممکن است از سیم و یا امواج استفاده شود. بر این اساس می‌توان شبکه‌ها را به دو دسته شبکه-های بیسیم و سیمی تقسیم نمود. در ادامه عمدتا منظور ما از شبکه، شبکه‌های سیمی است.

تقسیم بندی بر اساس حوزه جغرافی تحت پوشش:

شبکه های کامپیوتری با توجه به حوزه جغرافیائی تحت پوشش به سه گروه تقسیم می گردند :

- شبکه های محلی (کوچک) LAN
- شبکه های متوسط MAN
- شبکه های گسترده WAN

شبکه های LAN: حوزه جغرافیائی که توسط این نوع از شبکه ها پوشش داده می شود ، یک محیط کوچک نظیر یک ساختمان اداری است. این نوع از شبکه ها دارای ویژگی های زیر می باشند :

- توانائی ارسال اطلاعات با سرعت بالا
- محدودیت فاصله
- قابلیت استفاده از محیط مخابراتی ارزان نظیر خطوط تلفن به منظور ارسال اطلاعات
- نرخ پایین خطاء در ارسال اطلاعات با توجه به محدود بودن فاصله

شبکه های MAN : حوزه جغرافیائی که توسط این نوع شبکه ها پوشش داده می شود ، در حد و اندازه یک شهر و یا شهرستان است . ویژگی های این نوع از شبکه ها بشرح زیر است :

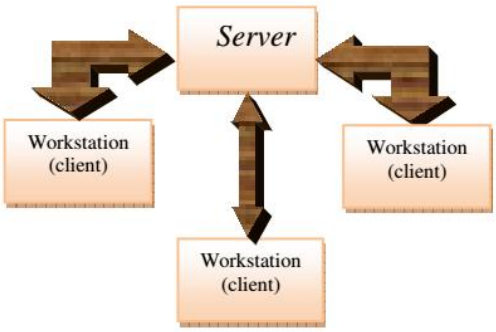
- پیچیدگی بیشتر نسبت به شبکه های محلی
- قابلیت ارسال تصاویر و صدا
- قابلیت ایجاد ارتباط بین چندین شبکه

شبکه های WAN : حوزه جغرافیائی که توسط این نوع شبکه ها پوشش داده می شود ، در حد و اندازه کشور و قاره است . ویژگی این نوع شبکه ها بشرح زیر است :

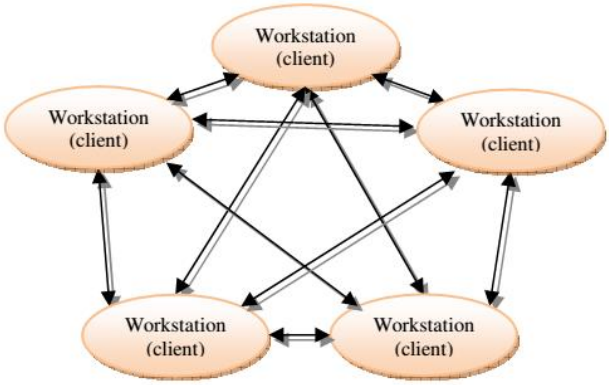
- قابلیت ارسال اطلاعات بین کشورها و قاره ها
- قابلیت ایجاد ارتباط بین شبکه های LAN
- سرعت پایین ارسال اطلاعات نسبت به شبکه های LAN
- نرخ خطای بالا با توجه به گستردگی محدوده تحت پوشش

تقسیم بندی بر اساس نوع وظایف

کامپیوترهای موجود در شبکه را با توجه به نوع وظایف مربوطه به دو گروه عمده : سرویس دهندگان (Servers) و یا سرویس گیرندگان (Clients) تقسیم می نمایند. کامپیوترهائی که در شبکه برای سایر کامپیوترها سرویس‌ها و خدماتی را ارائه می نمایند ، سرویس دهنده نامیده می‌گردند. کامپیوترهایی که از خدمات و سرویس های ارائه شده توسط سرویس دهندگان استفاده می کنند ، سرویس گیرنده نامیده می شوند . در شبکه های Client-Server ، یک کامپیوتر در شبکه نمی تواند هم به عنوان سرویس دهنده و هم به عنوان سرویس گیرنده ، ایفای وظیفه نماید.



البته در شبکه‌های Peer-To-Peer ، یک کامپیوتر می تواند هم بصورت سرویس دهنده و هم بصورت سرویس گیرنده ایفای وظیفه نماید.



یک شبکه LAN در ساده ترین حالت از اجزای زیر تشکیل شده است :

- **دو کامپیوتر شخصی**: یک شبکه می تواند شامل چند صد کامپیوتر باشد. حداقل یکی از کامپیوترها می بایست به عنوان سرویس دهنده مشخص گردد. (در صورتی که شبکه از نوع Client-Server باشد). سرویس دهنده، کامپیوتری است که هسته اساسی سیستم عامل بر روی آن نصب خواهد شد.
- **یک عدد کارت شبکه (NIC) برای هر دستگاه**: کارت شبکه نظیر کارت هائی است که برای مودم و صدا در کامپیوتر استفاده می گردد. کارت شبکه مسئول دریافت ، انتقال ، سازماندهی و ذخیره سازی موقت اطلاعات در طول شبکه است . به منظور انجام وظایف فوق کارت های شبکه دارای پردازنده، حافظه و گذرگاه اختصاصی خود هستند.

تقسیم بندی بر اساس توپولوژی

الگوی هندسی استفاده شده جهت اتصال کامپیوترها ، توپولوژی نامیده می شود. توپولوژی انتخاب شده برای پیاده سازی شبکه ها، عاملی مهم در جهت کشف و برطرف نمودن خطا در شبکه خواهد بود. انتخاب یک توپولوژی خاص نمی تواند بدون ارتباط با محیط انتقال و روش های استفاده از خط مطرح گردد. نوع توپولوژی انتخابی جهت اتصال کامپیوترها به یکدیگر، مستقیما بر نوع محیط انتقال و روش های استفاده از خط تاثیر می گذارد. با توجه به تاثیر مستقیم توپولوژی انتخابی در نوع کابل کشی و هزینه های مربوط به آن، می بایست با دقت و تامل به انتخاب توپولوژی یک شبکه همت گماشت. عوامل مختلفی جهت انتخاب یک توپولوژی بهینه مطرح می شود. مهمترین این عوامل بشرح ذیل است :

- **هزینه:** هزینه نصب شبکه می‌تواند بر حسب نوع توپولوژی متفاوت باشد.

نوع کاربرد: گاهی اوقات یک کاربرد خاص از شبکه می‌تواند مستلزم استفاده از نوع خاصی شبکه گردد.

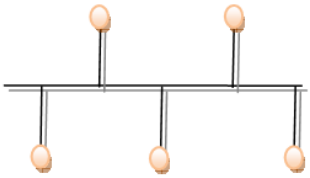
- **انعطاف پذیری:** یکی از مزایای شبکه‌های LAN، توانائی پردازش داده‌ها و گستردگی و توزیع گره‌ها در یک محیط است. بدین ترتیب توان محاسباتی سیستم و منابع موجود در اختیار تمام استفاده کنندگان قرار خواهد گرفت. توپولوژی انتخابی می بایست بسادگی امکان تغییر پیکربندی در شبکه را فراهم نماید. مثلا ایستگاهی را از نقطه‌ای به نقطه دیگر انتقال و یا قادر به ایجاد یک ایستگاه جدید در شبکه باشیم.

سه نوع توپولوژی رایج در شبکه های LAN استفاده می گردد:

- BUS
- STAR
- RING

توپولوژی BUS

یکی از رایجترین توپولوژی ها برای پیاده سازی شبکه های LAN است . در مدل فوق از یک کابل به عنوان ستون فقرات اصلی در شبکه استفاده شده و تمام کامپیوترهای موجود در شبکه (سرویس دهنده ، سرویس گیرنده) به آن متصل می گردند.



BUS

مزایای توپولوژی BUS

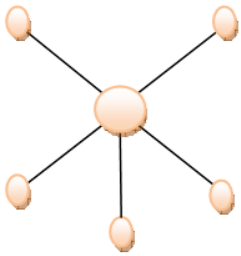
- **کم بودن طول کابل:** بدلیل استفاده از یک خط انتقال جهت اتصال تمام کامپیوترها، در توپولوژی فوق از کابل کمی استفاده می شود. موضوع فوق باعث پایین آمدن هزینه نصب و ایجاد تسهیلات لازم در جهت پشتیبانی شبکه خواهد بود.
- **ساختار ساده:** توپولوژی BUS دارای یک ساختار ساده است . در مدل فوق صرفا از یک کابل برای انتقال اطلاعات استفاده می شود.
- **توسعه آسان:** یک کامپیوتر جدید را می توان براحتی در نقطه ای از شبکه اضافه کرد. در صورت اضافه شدن ایستگاه‌های بیشتر در یک سگمنت، می توان از تقویت کننده هائی به نام Repeater استفاده کرد.

معایب توپولوژی BUS

- **مشکل بودن عیب یابی**: با اینکه سادگی موجود در تویولوژی BUS امکان بروز اشتباه را کاهش می دهند، ولی در صورت بروز خطاء کشف آن ساده نخواهد بود. در شبکه هائی که از توپولوژی فوق استفاده می نمایند ، کنترل شبکه در هر گره دارای مرکزیت نبوده و در صورت بروز خطاء می بایست نقاط زیادی به منظور تشخیص خطاء بازدید و بررسی گردند.
- **ایزوله کردن خطاء مشکل است**: در صورتی که یک کامپیوتر در توپولوژی فوق دچار مشکل گردد ، می بایست کامپیوتر را در محلی که به شبکه متصل است رفع عیب نمود. در موارد خاص می توان یک گره را از شبکه جدا کرد. در حالتیکه اشکال در محیط انتقال باشد ، تمام یک سگمنت می بایست از شبکه خارج گردد.
- **ماهیت تکرارکننده ها**: در مواردیکه برای توسعه شبکه از تکرارکننده ها استفاده می گردد، ممکن است در ساختار شبکه تغییراتی نیز داده شود. موضوع فوق مستلزم بکارگیری کابل بیشتر و اضافه نمودن اتصالات مخصوص شبکه است .

توپولوژی STAR

در این نوع توپولوژی همانگونه که از نام آن مشخص است، از مدلی شبیه "ستاره" استفاده می گردد. در این مدل تمام کامپیوترهای موجود در شبکه معمولا به یک دستگاه خاص با نام " هاب " متصل خواهند شد.



Star

مزایای توپولوژی STAR

- **سادگی سرویس شبکه:** توپولوژی STAR شامل تعدادی از نقاط اتصالی در یک نقطه مرکزی است . ویژگی فوق تغییر در ساختار و سرویس شبکه را آسان می نماید.
- **در هر اتصال یک دستگاه**: نقاط اتصالی در شبکه ذاتا مستعد اشکال هستند. در توپولوژی STAR اشکال در یک اتصال ، باعث خروج آن خط از شبکه و سرویس و اشکال زدائی خط مزبور است . عملیات فوق تاثیری در عملکرد سایر کامپیوترهای موجود در شبکه نخواهد گذاشت.
- **کنترل مرکزی و عیب یابی**: با توجه به این مسئله که نقطه مرکزی مستقیما به هر ایستگاه موجود در شبکه متصل است ، اشکالات و ایرادات در شبکه بسادگی تشخیص و مهار خواهند گردید.
- **روش های ساده دستیابی**: هر اتصال در شبکه شامل یک نقطه مرکزی و یک گره جانبی است. در چنین حالتی دستیابی به محیط انتقال جهت ارسال و دریافت اطلاعات دارای الگوریتمی ساده خواهد بود.

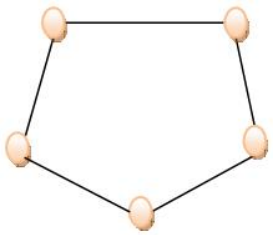
معایب توپولوژی STAR

- **زیاد بودن طول کابل**: بدلیل اتصال مستقیم هر گره به نقطه مرکزی ، مقدار زیادی کابل مصرف می شود. با توجه به اینکه هزینه کابل نسبت به تمام شبکه ، کم است ، تراکم در کانال کشی جهت کابل ها و مسائل مربوط به نصب و پشتیبنی آنها بطور قابل توجهی هزینه ها را افزایش خواهد داد.
- **مشکل بودن توسعه**: اضافه نمودن یک گره جدید به شبکه مستلزم یک اتصال از نقطه مرکزی به گره جدید است . با اینکه در زمان کابل کشی پیش بینی های لازم جهت توسعه در نظر گرفته می شود ، ولی در برخی حالات نظیر زمانیکه طول زیادی از کابل مورد نیاز بوده و یا اتصال مجموعه ای از گره های غیر قابل پیش بینی اولیه ، توسعه شبکه را با مشکل مواجه خواهد کرد.
- **وابستگی به نقطه مرکزی**: در صورتی که نقطه مرکزی (هاب) در شبکه با مشکل مواجه شود، تمام شبکه غیرقابل استفاده خواهد بود.

توپولوژی RING

در این نوع توپولوژی تمام کامپیوترها بصورت یک حلقه به یکدیگر مرتبط می گردند. تمام کامپیوترهای موجود در شبکه (سرویس دهنده، سرویس گیرنده) به یک کابل که بصورت یک دایره بسته است، متصل می گردند. در مدل فوق هر گره به دو

و فقط دو همسایه مجاور خود متصل است. اطلاعات از گره مجاور دریافت و به گره بعدی ارسال می شوند. بنابراین داده‌ها فقط در یک جهت حرکت کرده و از ایستگاهی به ایستگاه دیگر انتقال پیدا می کنند.



Ring

مزایای توپولوژی RING

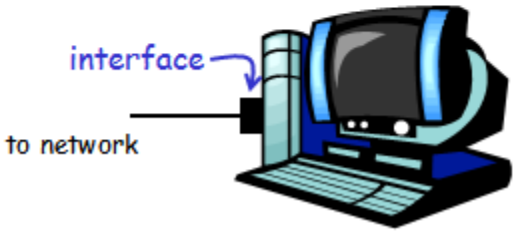
- **کم بودن طول کابل**: طول کابلی که در این مدل بکار گرفته می شود ، قابل مقایسه به توپولوژی BUS نبوده و طول کمی را در بردارد. ویژگی فوق باعث کاهش تعداد اتصالات (کانکتور) در شبکه شده و ضریب اعتماد به شبکه را افزایش خواهد داد.
- **نیاز به فضائی خاص جهت انشعابات در کابل کشی نخواهد بود:** بدلیل استفاده از یک کابل جهت اتصال هر گره به گره همسایه اش، اختصاص محل هائی خاص به منظور کابل کشی ضرورتی نخواهد داشت .
- **مناسب جهت فیبر نوری**: استفاده از فیبر نوری باعث بالا رفتن نرخ سرعت انتقال اطلاعات در شبکه است. چون در توپولوژی فوق ترافیک داده ها در یک جهت است، می توان از فیبر نوری به منظور محیط انتقال استفاده کرد.

معایب توپولوژی RING

- **اشکال در یک گره باعث اشکال در تمام شبکه می گردد**: در صورت بروز اشکال در یک گره ، تمام شبکه با اشکال مواجه خواهد شد. و تا زمانیکه گره معیوب از شبکه خارج نگردد ، هیچگونه ترافیک اطلاعاتی را روی شبکه نمی توان داشت .
- **اشکال زدائی مشکل است**: بروز اشکال در یک گره می تواند روی تمام گره‌های دیگر تاثیر گذار باشد. به منظور عیب یابی می بایست چندین گره بررسی شود تا گره مورد نظر پیدا گردد.
- **تغییر در ساختار شبکه مشکل است**: در زمان گسترش و یا اصلاح حوزه جغرافیائی تحت پوشش شبکه، بدلیل ماهیت حلقوی شبکه مسائلی بوجود خواهد آمد .
- **توپولوژی بر روی نوع دستیابی تاثیر می گذارد**: هر گره در شبکه دارای مسئولیت عبور دادن داده‌ای است که از گره مجاور دریافت داشته است. قبل از اینکه یک گره بتواند داده خود را ارسال نماید، می بایست به این اطمینان برسد که محیط انتقال برای استفاده قابل دستیابی است.

Addresses and Interfaces

- *interface:* connection between host or router and the physical network link
  - routers typically have multiple interfaces
  - hosts may have multiple interfaces
- Interfaces have addresses
  - Hosts don't have addresses(their interface does)
  - Routers don't have addresses (their interfaces do)



7

Internet addressing schemes

- A host interface has 3 types of addresses:
  - host name (Application Layer): e.g., medellin.cs.columbia.edu
  - IP address (Network Layer or Layer 3): e.g., 128.119.40.7
  - MAC address (Link Layer or Layer 2): e.g., E6-E9-00-17-BB-4B
- Actually, so do router interfaces:

traceroute cs.umass.edu (from medellin.cs.columbia.edu)

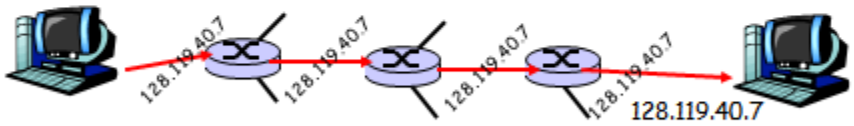
1 mudd-edge-1.net-columbia.edu (128.119.240.41)
2 nyser-gw.net.columbia.edu (128.59.16.1)
3 nn2k-gw.net.columbia.edu (128.59.1.6)
4 vbns-columbia1.nysernet.net (199.109.4.6)
5 jn1-at1-0-0-17.cht.vbns.net (204.147.132.130)
etc..

8

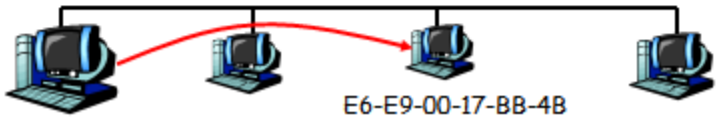
Why 3 Addressing Schemes?

- host names: convenient app-to-app communication

- IP: efficient large-scale network communication

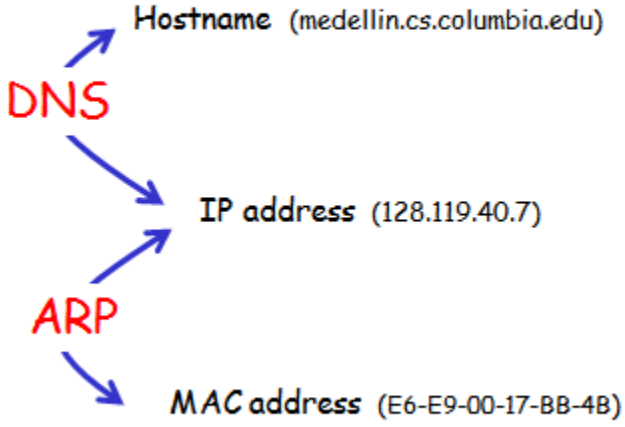


- MAC: quick-n-easy LAN forwarding



9

Translating between addresses



10

DNS: Domain Name System

People: many identifiers:
- SSN, name, Passport #

Internet hosts, routers:
- IP address (32 bit) - used for addressing datagrams
- "name", e.g., gaia.cs.umass.edu - used by humans

Domain Name System:
- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol* host, routers, name servers to communicate to *resolve* names (address/name translation)
  - note: core Internet function implemented as application-layer protocol
  - complexity at network's "edge" – interior routers don't maintain any DNS-related info

11

DNS name servers

Why not centralize DNS?
- single point of failure
- traffic volume
- distant centralized database
- maintenance

doesn't *scale!*

- no server has all name-to-IP address mappings

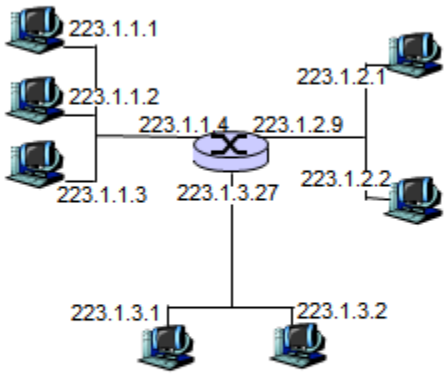
local name servers:
- each ISP, company has *local (default) name server*
- host DNS query first goes to local name server

authoritative name server:
- for a host: stores that host's IP address, name
- can perform name/address translation for that host's name

12

IP Addressing

- IP address: 32-bit identifier for host, router *interface*
  - IP addresses associated with interface, not host, router



- DHCP: Dynamic Host Configuration Protocol
  - some IP addresses left open
  - can be dynamically assigned (e.g., to a laptop) when interface connected

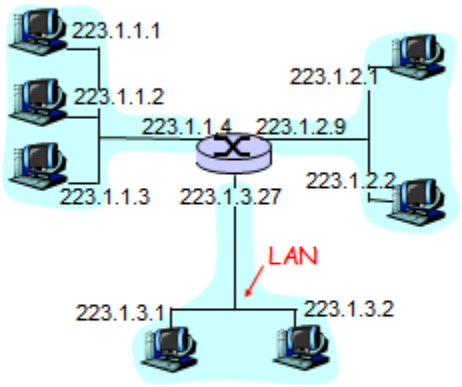
223.1.1.1 = 11011111 00000001 00000001 00000001

223 1 1 1

18

IP Addressing

- IP address:
  - network part (high order bits)
  - host part (low order bits)
- *What's a network ?* (from IP address perspective)
  - device interfaces with same network part of IP address
  - can physically reach each other without intervening router (i.e., on the same LAN)



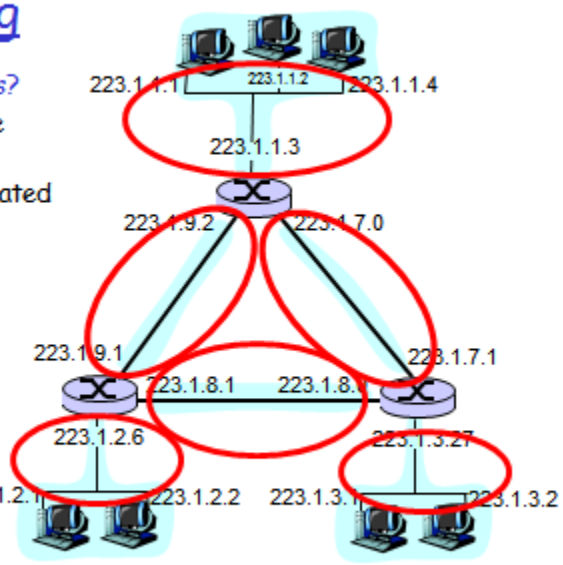
network consisting of 3 IP networks (for IP addresses starting with 223, first 24 bits are network address)

19

IP Addressing

How to find the networks?

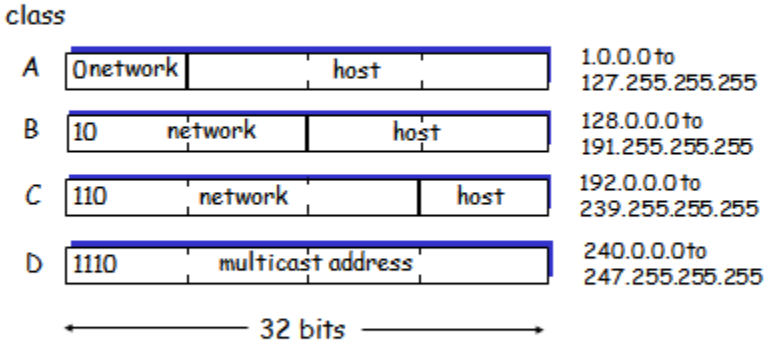
- Detach each interface from router, host
- create "islands of isolated networks"



Interconnected system consisting of six networks

20

IP Addresses: Class-based (Old)



21

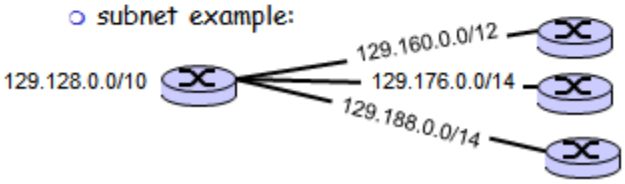
CIDR addressing (New)

- Classless Interdomain Routing
  - network part can be any # of bits
  - Format: a.b.c.d/x, where x indicates # of bits in network part (the prefix)

18 relevant bits

128.119.48.12/18 = 10000000 01110111 00110000 00001100

  - high order bits form the **prefix**
  - once inside the network, can **subnet**: divide remaining 24-x bits
  - subnet example:

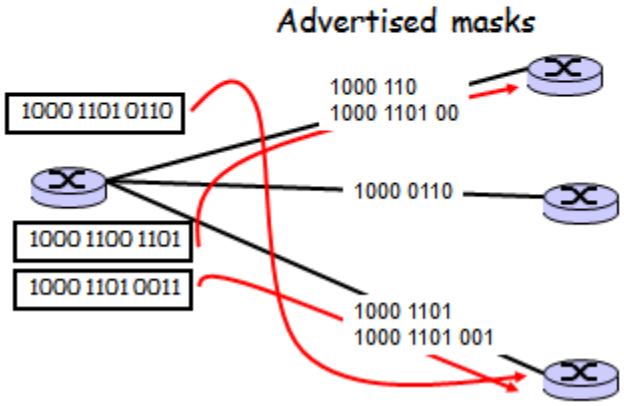


Note: picture shows prefix masks, not interface addrs!

22

Routing with CIDR

- Packet should be sent toward the interface with the **longest matching prefix**



23

Hierarchical Routing

Our routing study thus far - idealization
all routers identical
network "flat"
... *not* true in practice

scale: with 50 million destinations:

- can't store all dest's in routing tables!
- routing table exchange would swamp links!

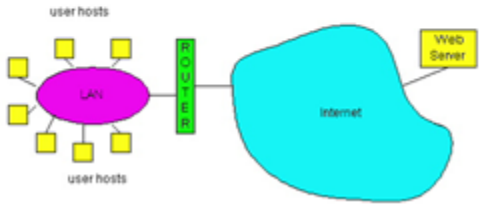
administrative autonomy

- internet = network of networks
- each network admin may want to control routing in its own network

24

LAN technologies (Link Layer)

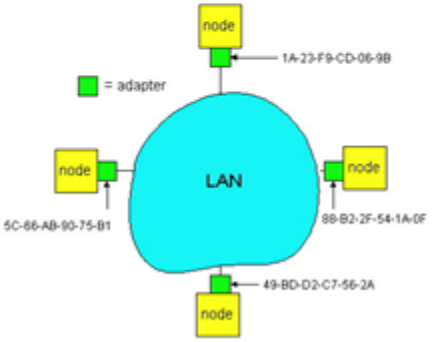
- MAC protocols used in LANs, to control access to the channel
- **Token Rings:** IEEE 802.5 (IBM token ring), for computer room, or Department connectivity, up to 16Mbps; FDDI (Fiber Distributed Data Interface), for Campus and Metro connectivity, up to 200 stations, at 100Mbps.
- **Ethernets:** employ the CSMA/CD protocol; 10Mbps (IEEE 802.3), Fast E-net (100Mbps), Giga E-net (1,000 Mbps); by far the most popular LAN technology



29

LAN Addresses and ARP

- **IP address**: drives the packet to destination **network**
- **LAN** (or MAC or Physical) **address**: drives the packet to the destination node's LAN interface card (adapter card) on the **local LAN**
- **48 bit MAC address** (for most LANs); burned in the adapter ROM
  - the address stays with the card
  - card's MAC address can't be changed



30

LAN Address (more)

- MAC address allocation administered by IEEE
- A manufacturer buys a portion of the address space (to assure uniqueness)
- Analogy:
  (a) MAC address: like Social Security Number
  (b) IP address: like postal address

- MAC flat address => portability
- IP hierarchical address NOT portable (address stays with the network, not the host interface)
- Broadcast LAN address: 1111……1111

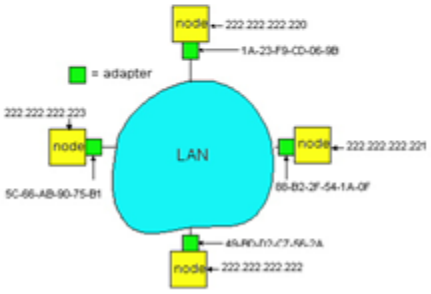
31

ARP: Address Resolution Protocol

- MAC address ↔ IP address
- Each IP node (Host, Router) on the LAN has **ARP** module and Table
- ARP Table: IP/MAC address mappings for **some** LAN nodes

  < IP address; MAC address; TTL>

  < ..... >
- TTL (Time To Live): timer, typically 20 min



32

ARP (more)

- Host A wants to send packet to destination IP addr XYZ on same LAN
- Source Host first checks own ARP Table for IP addr XYZ
- If XYZ **not** in the ARP Table, ARP module **broadcasts** ARP pkt:

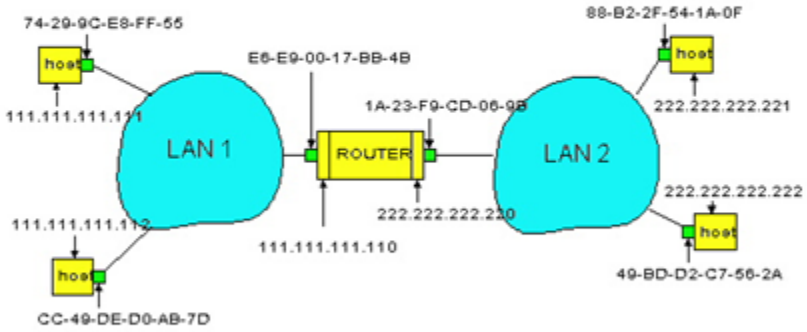
  < XYZ, MAC (?) >
- ALL nodes on the LAN accept and inspect the ARP pkt
- Node XYZ responds with **unicast** ARP pkt carrying own MAC addr:

  < XYZ, MAC (XYZ) >
- MAC address **cached** in ARP Table
- Benefit of ARP: self-configuring (plug-n-play) – makes life easier for the sys-admin!!

33

Routing pkt to another LAN

- Say, route packet from source IP addr <111.111.111.111> to destination addr <222.222.222.222>



- In routing table at source Host, find router 111.111.111.110
- In ARP table at source, find MAC address E6-E9-00-17-BB-4B, etc

34

Ethernet Frame Structure

- Sending adapter encapsulates an IP datagram (or other network layer protocol packet) in **Ethernet Frame** which contains a Preamble, a Header, Data, and CRC fields
- **Preamble**: 7 bytes with the pattern 10101010 followed by one byte with the pattern 10101011; used for synchronizing receiver to sender clock (clocks are never exact, some drift is highly likely)

36

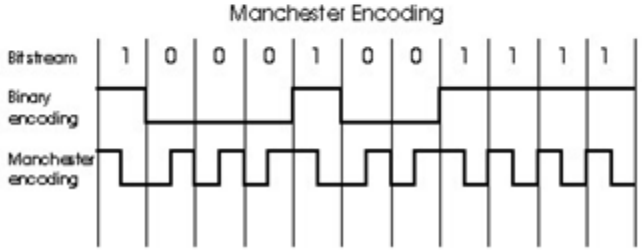
Ethernet Frame Structure (more)

- Header contains Destination and Source Addresses and a Type field
- **Addresses**: 6 bytes, frame is received by all adapters on a LAN and dropped if address does not match
- **Type**: indicates the higher layer protocol, mostly IP but others may be supported such as Novell IPX and AppleTalk)
- **CRC**: checked at receiver, if error is detected, the frame is simply dropped

37

Baseband Manchester Encoding

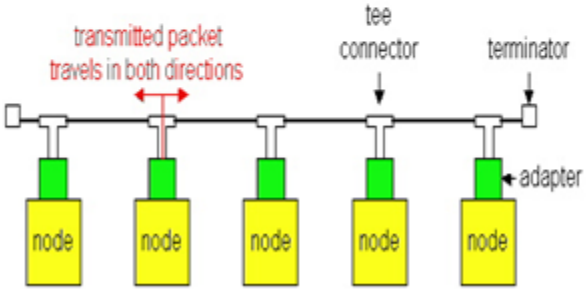
- Baseband here means that no carrier is modulated; instead bits are encoded using Manchester encoding and transmitted directly by modified voltage of a DC signal
- Manchester encoding ensures that a voltage transition occurs in each bit time which helps with receiver and sender clock synchronization



38

Ethernet Technologies: 10Base2

- 10==10Mbps; 2==under 200 meters maximum length of a cable segment; also referred to as "Cheapnet"
- Uses thin coaxial cable in a bus topology
- Repeaters are used to connect multiple segments (up to 5); a repeater repeats the bits it hears on one interface to its other interfaces, ie a physical layer device only!



39

Hubs, Bridges, and Switches

- Used for extending LANs in terms of geographical coverage, number of nodes, administration capabilities, etc.
- Differ in regards to:
  - collision domain isolation
  - layer at which they operate
- Different than routers
  - hubs, bridges, and switches are plug and play
  - don't provide optimal routing of IP packets

40

Hubs

- Physical Layer devices: essentially repeaters operating at bit levels: repeat received bits on one interface to all other interfaces
- Hubs can be arranged in a hierarchy (or **multi-tier design**), with a **backbone** hub at its top

41

Hubs (more)

- Each connected LAN is referred to as a LAN **segment**
- Hubs **do not isolate** collision domains: a node may collide with any node residing at any segment in the LAN

- Hub Advantages:
  - Simple, inexpensive device
  - Multi-tier provides graceful degradation: portions of the LAN continue to operate if one of the hubs malfunction
  - Extends maximum distance between node pairs (100m per Hub)
  - can disconnect a "jabbering adapter"; 10base2 would not work if an adapter does not stop transmitting on the cable
  - can gather monitoring information and statistics for display to LAN administrators

42

Hubs (more)

- Hub Limitations:
  - Always broadcasts pkts (i.e., no smarts about which link to send on)
  - Single collision domain results in no increase in max throughput; the multi-tier throughput same as the the single segment throughput
  - Individual LAN restrictions pose limits on the number of nodes in the same collision domain (thus, per Hub); and on the total allowed geographical coverage
  - Cannot connect different Ethernet types (e.g., 10BaseT and 100baseT)

43

10BaseT and 100BaseT

- 10/100 Mbps rate; latter called "fast ethernet"
- T stands for Twisted Pair
- 10BaseT and 100BaseT use Hubs

44

10BaseT and 100BaseT (more)

- Max distance from node to Hub is 100 meters
- 100BaseT does not use Manchester encoding; it uses 4B5B for better coding efficiency

45

Bridges

- **Link Layer devices**: they operate on Ethernet frames, examining the frame header and selectively forwarding a frame base on its destination
- Bridge **isolates collision** domains since it buffers frames
- When a frame is to be forwarded on a segment, the bridge uses CSMA/CD to access the segment and transmit
- Are also self-configuring (plug-n-play)

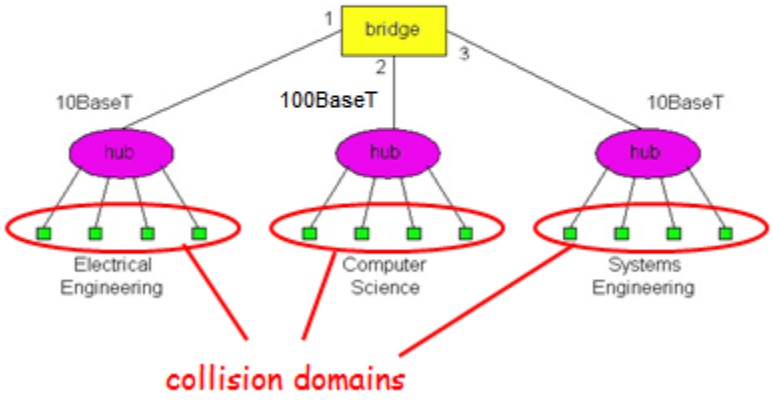
46

Bridges (more)

- **Bridge advantages:**
  - Isolates collision domains resulting in higher total max throughput, and does not limit the number of nodes nor geographical coverage
  - Can connect different type Ethernet since it is a store and forward device
  - Transparent: no need for any change to hosts LAN adapters

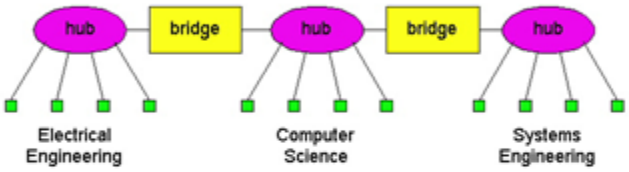
47

Backbone Bridge



48

Interconnection Without Backbone



- **Not recommended** for two reasons:
  - Single point of failure at Computer Science hub
  - All traffic between EE and SE must path over CS segment

49

Bridge Filtering

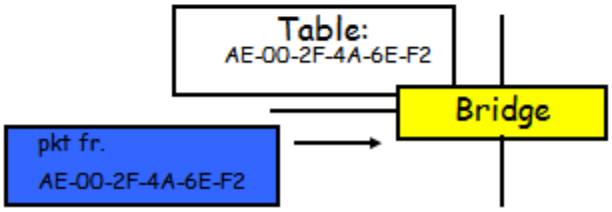
- Bridges learn which hosts can be reached through which interfaces and maintain filtering tables
- A filtering table entry:
  (Node LAN Address, Bridge Interface, Time Stamp)
- Filtering procedure:

```
if destination is on LAN on which frame was received
    then drop the frame
    else { lookup filtering table
           if entry found for destination
               then forward the frame on interface indicated;
               else flood;   /* forward on all but the interface on
                                which the frame arrived*/
         }
```

50

Bridge Learning

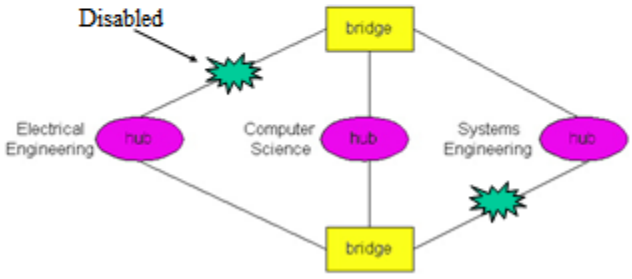
- When a frame is received, the bridge "learns" from the source address and updates its filtering table (Node LAN Address, Bridge Interface, Time Stamp)
- Stale entries in the Filtering Table are dropped (TTL can be 60 minutes)



51

Bridges Spanning Tree

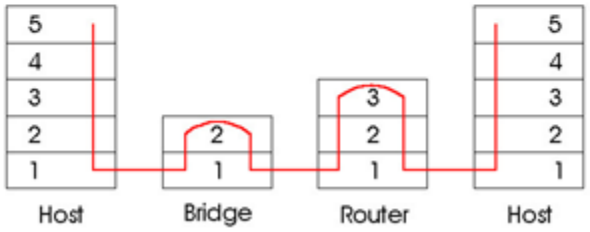
- For increased reliability, it is desirable to have redundant, alternate paths from a source to a destination
- With multiple simultaneous paths however, cycles result on which bridges may multiply and forward a frame forever
- Solution is organizing the set of bridges in a spanning tree by disabling a subset of the interfaces in the bridges:



52

Bridges vs. Routers

- Both are store-and-forward devices, but Routers are Network Layer devices (examine network layer headers) and Bridges are Link Layer devices
- Routers maintain routing tables and implement routing algorithms, bridges maintain filtering tables and implement filtering, learning and spanning tree algorithms



53

Routers vs. Bridges

- Bridges + and -

+ Bridge operation is simpler requiring less processing bandwidth

- Topologies are restricted with bridges: a spanning tree must be built to avoid cycles

- Bridges do not offer protection from broadcast storms (endless broadcasting by a host will be forwarded by a bridge: cost of plug-n-play)

54

Routers vs. Bridges

- Routers + and -

+ Arbitrary topologies can be supported, cycling is limited by TTL counters (and good routing protocols)
+ Provide firewall protection against broadcast storms
- Require IP address configuration (not plug and play)
- Require higher processing bandwidth

- Bridges do well in small (few hundred hosts) while routers are required in large networks (thousands of hosts)

55

Ethernet Switches

- A switch is a device that incorporates bridge functions as well as point-to-point 'dedicated connections'
- A host attached to a switch via a dedicated point-to-point connection; will always sense the medium as idle; no collisions ever!
- Ethernet Switches provide a combinations of shared/dedicated, 10/100/1000 Mbps connections

56

Ethernet

- Some E-net switches support cut-through switching: frame forwarded immediately to destination without awaiting for assembly of the entire frame in the switch buffer; slight reduction in latency
- Ethernet switches vary in size, with the largest ones incorporating a high bandwidth interconnection network

57

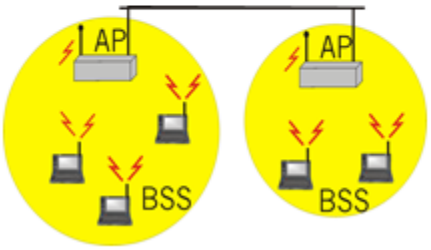
Hardware in the Layering Hierarchy

Network	Routers
Link	Bridges, Switches
Physical	Repeaters, Hubs

60

IEEE 802.11 Wireless LAN

- Wireless LANs are becoming popular for mobile Internet access
- Applications: nomadic Internet access, portable computing, ad hoc networking (multihopping)
- IEEE 802.11 standards defines MAC protocol; unlicensed frequency spectrum bands: 900Mhz, 2.4Ghz
- **Basic Service Sets + Access Points => Distribution System**
- Like a bridged LAN (flat MAC address)



61

Ad Hoc Networks

- IEEE 802.11 stations can dynamically form a group without AP
- Ad Hoc Network: no pre-existing infrastructure
- Applications: "laptop" meeting in conference room, car, airport; interconnection of "personal" devices (see bluetooth.com); battelfield; pervasive computing (smart spaces)
- IETF MANET (Mobile Ad hoc Networks) working group

62

PPP: Point to point protocol

- LAN-like connectivity for a host (e.g., over a modem-line)
  - (when used w/ IP, assigns an IP address to the host
- Pkt framing: encapsulation of packets
- bit transparency: must carry any bit pattern in the data field
- error detection (no correction)
- multiple network layer protocols
- connection liveness
- Network Layer Address negotiation: Hosts/nodes across the link must learn/configure each other's network address

63

Not Provided by PPP

- error correction/recovery
- flow control
- sequencing
- multipoint links (e.g., polling)

64

مروری بر TCP

TCP: Overview

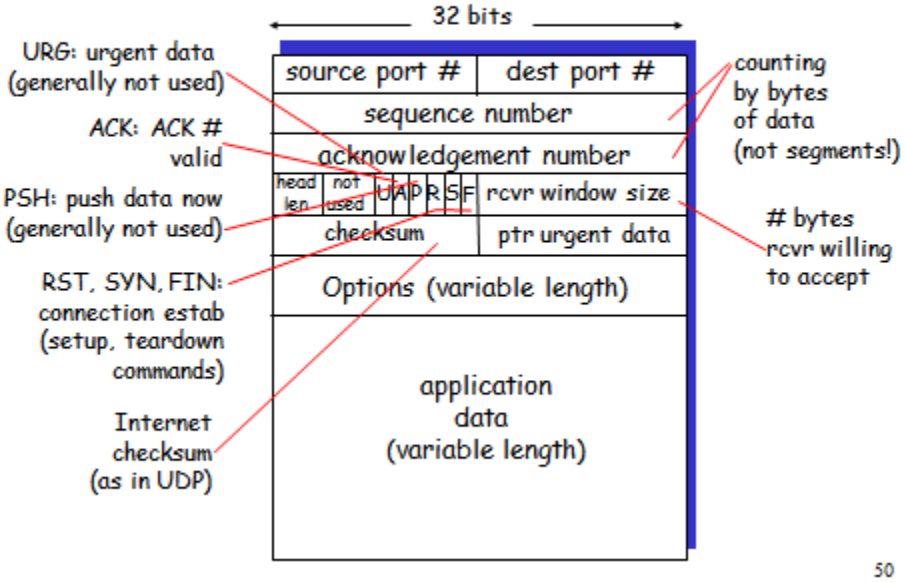
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

49

TCP segment structure



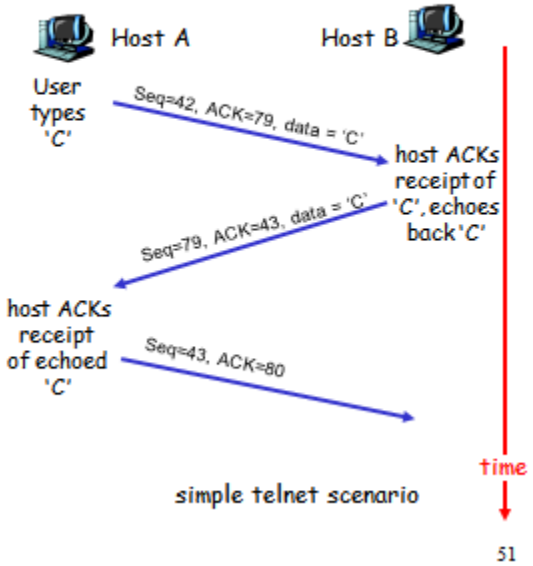
50

TCP seq. #'s and ACKs

Seq. #'s:
- byte stream "number" of first byte in segment's data

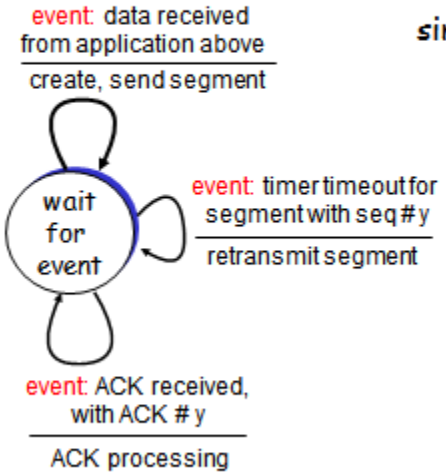
ACKs:
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor



51

TCP: reliable data transfer



simplified sender, assuming

- one way data transfer
- no flow, congestion control

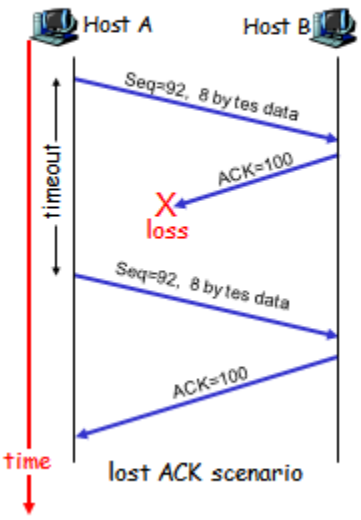
52

TCP ACK generation [RFC 1122, RFC 2581]

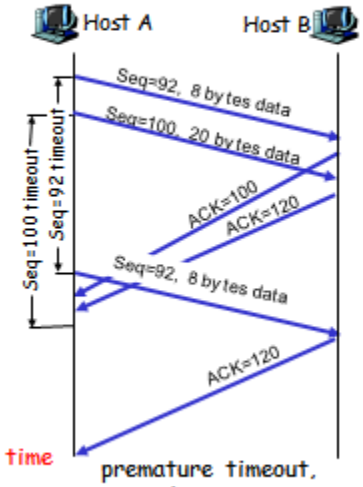
Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

54

TCP: retransmission scenarios



lost ACK scenario



premature timeout, cumulative ACKs

55

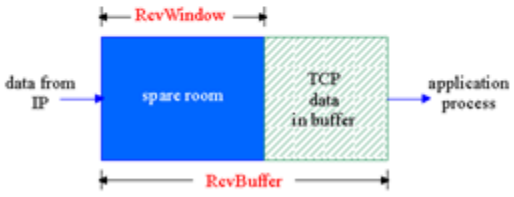
TCP Flow Control

flow control

sender won't overrun receiver's buffers by transmitting too much, too fast

RcvBuffer = size of TCP Receive Buffer

RcvWindow = amount of spare room in Buffer



receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- RcvWindow **field** in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received RcvWindow

56

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. RcvWindow)
- *client:* connection initiator

```
Socket clientSocket = new
Socket("hostname","port
number");
```

- *server:* contacted by client

```
Socket connectionSocket =
welcomeSocket.accept();
```

Three way handshake:

Step 1: client end system sends TCP SYN control segment to server
- specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment
- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

Initial seqnos in both directions chosen randomly. Why?

59

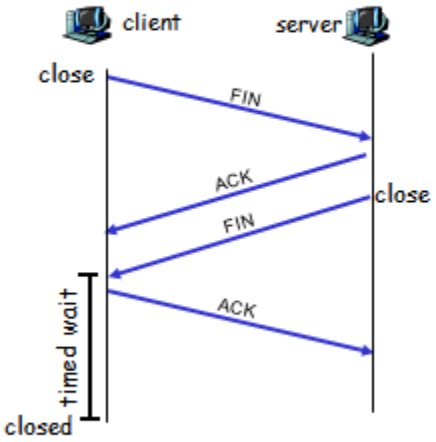
TCP Connection Management (cont.)

Closing a connection:

client closes socket:
`clientSocket.close();`

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



60

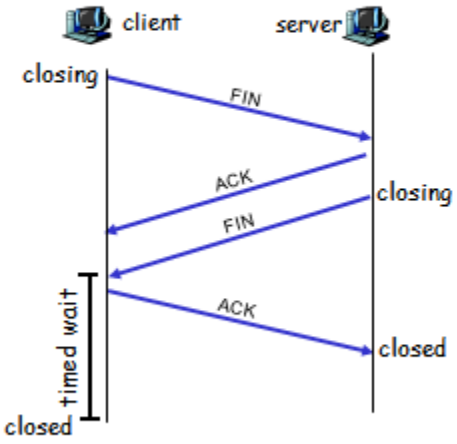
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



61

آشنایی با برخی دستورات در لینوکس مرتبط با شکبه

NETWORKING

PING

Utility usually used to check if communication can be made with another host. Can be used with default settings by just specifying a hostname (e.g. `ping raspberrypi.org`) or an IP address (e.g. `ping 8.8.8.8`). Can specify the number of packets to send with the `-c` flag.

NMAP

Network exploration and scanning tool. Can return port and OS information about a host or a range of hosts. Running just `nmap` will display the options available as well as example usage.

HOSTNAME

Displays the current hostname of the system. A privileged (super) user can set the hostname to a new one by supplying it as an argument (e.g. `hostname new-host`).

IFCONFIG

Displays the network configuration details for the interfaces on the current system when run without any arguments (i.e. `ifconfig`). By supplying the command with the name of an interface (e.g. `eth0` or `lo`) you can then alter the configuration (check the man-page for more details).

آشنایی با socket programming

Socket Programming

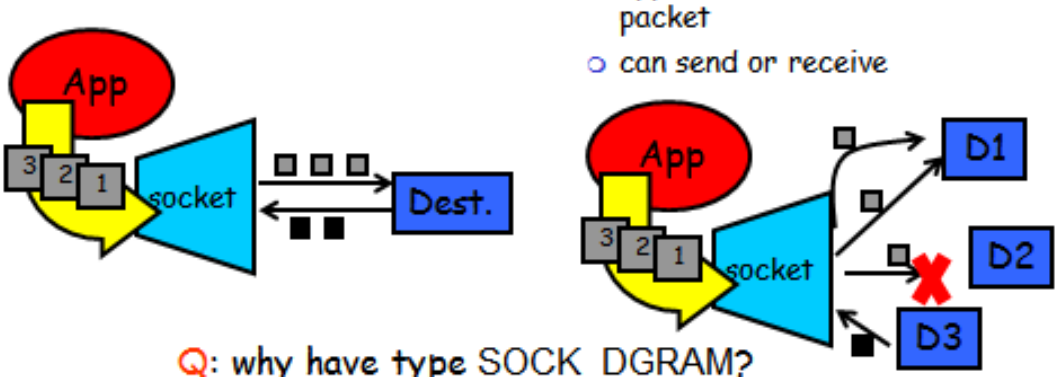
- What is a socket?
- Using sockets
  - Types (Protocols)
  - Associated functions
  - Styles

What is a socket?

- An interface between application and network
  - The application creates a socket
  - The socket *type* dictates the style of communication
    - reliable vs. best effort
    - connection-oriented vs. connectionless
- Once configured the application can
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)

Two essential types of sockets

- SOCK_STREAM
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional
- SOCK_DGRAM
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of "connection" – app indicates dest. for each packet
  - can send or receive



Q: why have type SOCK_DGRAM?

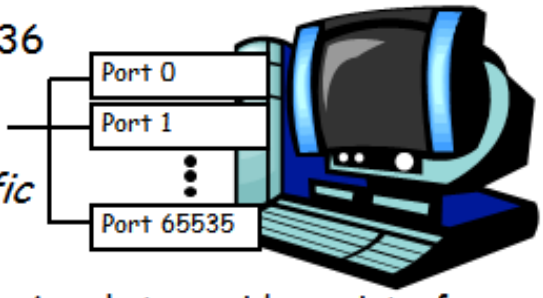
Socket Creation in C: socket

- int s = socket(domain, type, protocol);
  - s: socket descriptor, an integer (like a file-handle)
  - domain: integer, communication domain
    - e.g., PF_INET (IPv4 protocol) – typically used
  - type: communication type
    - SOCK_STREAM: reliable, 2-way, connection-based service
    - SOCK_DGRAM: unreliable, connectionless,
    - other values: need root permission, rarely used, or obsolete
  - protocol: specifies protocol (see file /etc/protocols for a list of options) - usually set to 0
- NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

4

Ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



- A socket provides an interface to send data to/from the network through a port

6

Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)
- Q: How do you choose which port a socket connects to?

7

The bind function

- associates and (can exclusively) reserves a port for use by the socket
- int status = bind(sockid, &addrport, size);
  - status: error status, = -1 if bind failed
  - sockid: integer, socket descriptor
  - addrport: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY - chooses a local address)
  - size: the size (in bytes) of the addrport structure
- bind can be skipped for both types of sockets. When and why?

8

Skipping the bind

- SOCK_DGRAM:
  - if only sending, no need to bind. The OS finds a port each time the socket sends a pkt
  - if receiving, need to bind
- SOCK_STREAM:
  - destination determined during conn. setup
  - don't need to know port sending from (during connection setup, receiving end is informed of port)

9

Connection Setup (SOCK_STREAM)

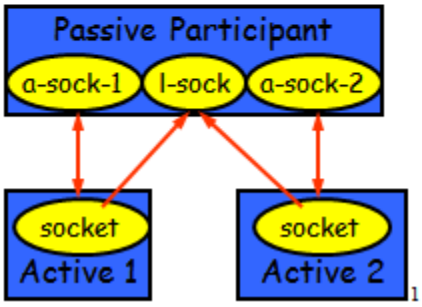
- Recall: no connection setup for SOCK_DGRAM
- A connection occurs between two kinds of participants
  - passive: waits for an active participant to request connection
  - active: initiates connection request to passive side
- Once connection is established, passive and active participants are "similar"
  - both can send & receive data
  - either can terminate the connection

10

Connection setup cont'd

- Passive participant
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants
- Why?

- Active participant
  - step 2: request & establish connection
  - step 4: data transfer



1

Connection setup: listen & accept

- Called by passive participant
- int status = listen(sock, queuelen);
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can "wait" for a connection
  - listen is **non-blocking**: returns immediately
- int s = accept(sock, &name, &namelen);
  - s: integer, the new socket (used for data-transfer)
  - sock: integer, the orig. socket (being listened on)
  - name: struct sockaddr, address of the active participant
  - namelen: sizeof(name): value/result parameter
    - must be set appropriately before call
    - adjusted by OS upon return
  - accept is **blocking**: waits for connection before returning

12

connect call

- int status = connect(sock, &name, namelen);
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - name: struct sockaddr: address of passive participant
  - namelen: integer, sizeof(name)
- connect is **blocking**

13

Sending / Receiving Data

- With a connection (SOCK_STREAM):
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]

14

Sending / Receiving Data (cont'd)

- Without a connection (SOCK_DGRAM):
  - int count = sendto(sock, &buf, len, flags, &addr, addrlen);
    - count, sock, buf, len, flags: same as send
    - addr: struct sockaddr, address of the destination
    - addrlen: sizeof(addr)
  - int count = recvfrom(sock, &buf, len, flags, &addr, &addrlen);
    - count, sock, buf, len, flags: same as recv
    - name: struct sockaddr, address of the source
    - namelen: sizeof(name): value/result parameter
- Calls are **blocking** [returns only after data is sent (to socket buf) / received]

15

close

- When finished using a socket, the socket should be closed:
- status = close(s);
  - status: 0 if successful, -1 if error
  - s: the file descriptor (socket being closed)
- Closing a socket
  - closes a connection (for SOCK_STREAM)
  - frees up the port used by the socket

16

The struct sockaddr

- The generic:

```
struct sockaddr {
    u_short sa_family;
    char sa_data[14];
};
```

  - sa_family
    - specifies which address family is being used
    - determines how the remaining 14 bytes are used

- The Internet-specific:

```
struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

  - sin_family = AF_INET
  - sin_port: port # (0-65535)
  - sin_addr: IP-address
  - sin_zero: unused

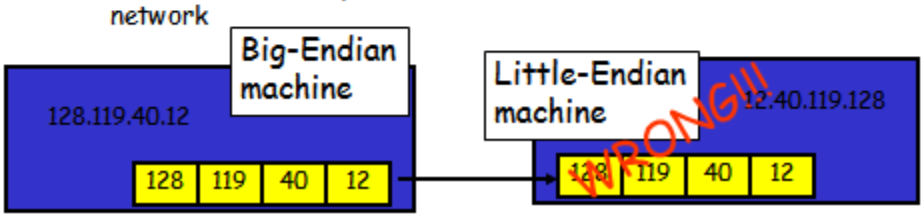
17

Address and port byte-ordering

- Address and port are stored as integers
  - u_short sin_port; (16 bit)
  - in_addr sin_addr; (32 bit)

```
struct in_addr {
  u_long s_addr;
};
```

- Problem:
  - different machines / OS's use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - these machines may communicate with one another over the network



Solution: Network Byte-Ordering

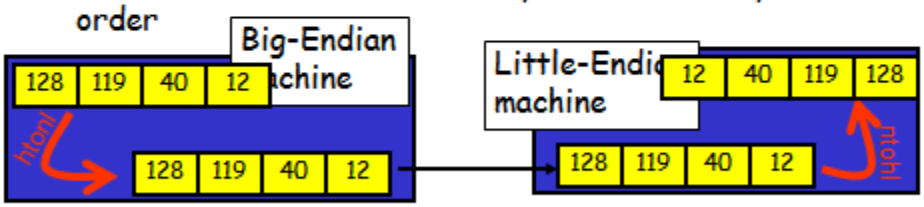
- Defs:
  - Host Byte-Ordering: the byte ordering used by a host (big or little)
  - Network Byte-Ordering: the byte ordering used by the network – always big-endian
- Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)
- Q: should the socket perform the conversion automatically?
- Q: Given big-endian machines don't need conversion routines and little-endian machines do, how do we avoid writing two versions of code?

19

UNIX's byte-ordering funcs

- u_long htonl(u_long x);
- u_short htons(u_short x);
- u_long ntohl(u_long x);
- u_short ntohs(u_short x);

- On big-endian machines, these routines do nothing
- On little-endian machines, they reverse the byte order



- Same code would have worked regardless of endian-ness of the two machines

20

Dealing with blocking calls

- Many of the functions we saw block until a certain event
  - accept: until a connection comes in
  - connect: until the connection is established
  - recv, recvfrom: until a packet (of data) is received
  - send, sendto: until data is pushed into socket's buffer
    - Q: why not until received?
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

21

Dealing w/ blocking (cont'd)

- **Options:**
  - create multi-process or multi-threaded code
  - turn off the blocking feature (e.g., using the fcntl file-descriptor control function)
  - use the select function call.
- **What does select do?**
  - can be permanent blocking, time-limited blocking or non-blocking
  - input: a set of file-descriptors
  - output: info on the file-descriptors' status
  - i.e., can identify sockets that are "ready for use": calls involving that socket will return immediately

22

select function call

- int status = select(nfds, &readfds, &writefds, &exceptfds, &timeout);
  - status: # of ready objects, -1 if error
  - nfds: 1 + largest file descriptor to check
  - readfds: list of descriptors to check if read-ready
  - writefds: list of descriptors to check if write-ready
  - exceptfds: list of descriptors to check if an exception is registered
  - timeout: time after which select returns, even if nothing ready - can be 0 or ∞
    (point timeout parameter to NULL for ∞)

23

To be used with select:

- Recall select uses a structure, struct fd_set
  - it is just a bit-vector
  - if bit *i* is set in [readfds, writefds, exceptfds], select will check if file descriptor (i.e. socket) *i* is ready for [reading, writing, exception]
- Before calling select:
  - FD_ZERO(&fdvar): clears the structure
  - FD_SET(i, &fdvar): to check file desc. *i*
- After calling select:
  - int FD_ISSET(i, &fdvar): boolean returns TRUE iff *i* is "ready"

24

Other useful functions

- bzero(char* c, int n): 0's n bytes starting at c
- gethostname(char *name, int len): gets the name of the current host
- gethostbyaddr(char *addr, int len, int type): converts IP hostname to structure containing long integer
- inet_addr(const char *cp): converts dotted-decimal char-string to long integer
- inet_ntoa(const struct in_addr in): converts long to dotted-decimal notation

- Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order

25

Release of ports

- Sometimes, a "rough" exit from a program (e.g., ctrl-c) does not properly free up a port
- Eventually (after a few minutes), the port will be freed
- To reduce the likelihood of this problem, include the following code:

```
#include <signal.h>
void cleanExit(){exit(0);}
```

  - in socket code:

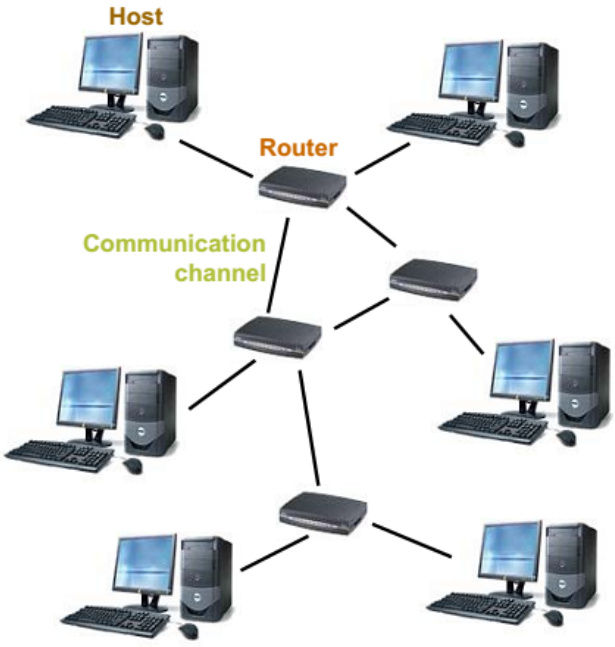
```
signal(SIGTERM, cleanExit);
signal(SIGINT, cleanExit);
```

26

Introduction

- Computer Network
  - hosts, routers, communication channels
- **Hosts** run applications
- **Routers** forward information
- **Packets**: sequence of bytes
  - contain control information
  - e.g. destination host
- **Protocol** is an agreement
  - meaning of packets
  - structure and size of packets

  e.g. Hypertext Transfer Protocol (HTTP)



Protocol Families - TCP/IP

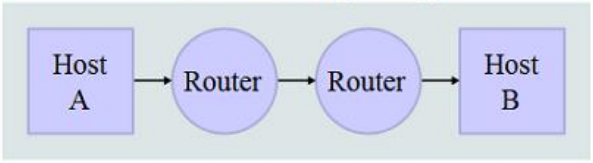
- Several protocols for different problems

☞ **Protocol Suites** or **Protocol Families:** TCP/IP

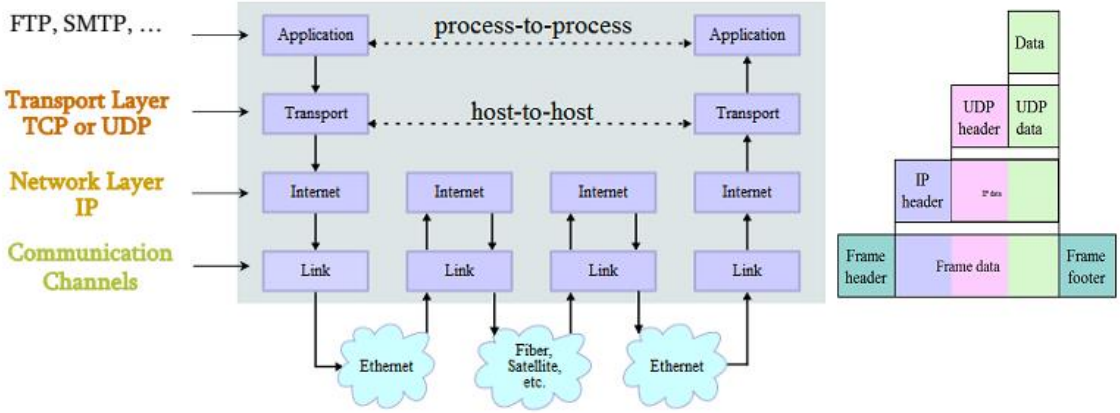
- TCP/IP provides **end-to-end** connectivity specifying how data should be
  - formatted,
  - addressed,
  - transmitted,
  - routed, and
  - received at the destination
- can be used in the internet and in stand-alone private networks
- it is organized into **layers**

TCP/IP

Network Topology *



Data Flow



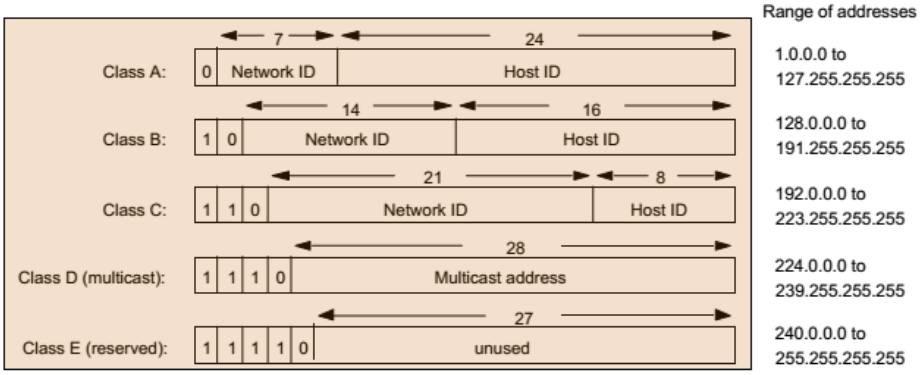
Internet Protocol (IP)

- provides a **datagram** service
  - packets are handled and delivered independently
- **best-effort** protocol
  - may loose, reorder or duplicate packets

- each packet must contain an **IP address** of its destination

Addresses - IPv4

- The **32** bits of an IPv4 address are broken into **4 octets**, or 8 bit fields (0-255 value in decimal notation).
- For networks of different size,
  - the first one (for large networks) to three (for small networks) octets can be used to identify the **network**, while
  - the rest of the octets can be used to identify the **node** on the network.



TCP vs UDP

- Both use **port numbers**
  - application-specific construct serving as a communication endpoint
  - 16-bit unsigned integer, thus ranging from 0 to 65535
  - ☞ to provide **end-to-end** transport
- UDP: User Datagram Protocol
  - no acknowledgements
  - no retransmissions
  - out of order, duplicates possible
  - connectionless, i.e., app indicates destination for each packet
- TCP: Transmission Control Protocol
  - reliable **byte-stream channel** (in order, all arrive, no duplicates)
    - similar to file I/O
  - flow control
  - connection-oriented
  - bidirectional

TCP vs UDP

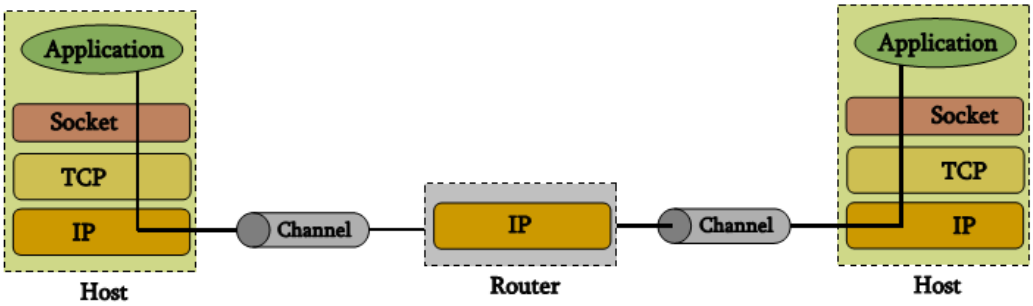
- TCP is used for services with a large data capacity, and a persistent connection
- UDP is more commonly used for quick lookups, and single use query-reply actions.

- Some common examples of TCP and UDP with their default ports:

DNS lookup	UDP	53
FTP	TCP	21
HTTP	TCP	80
POP3	TCP	110
Telnet	TCP	23

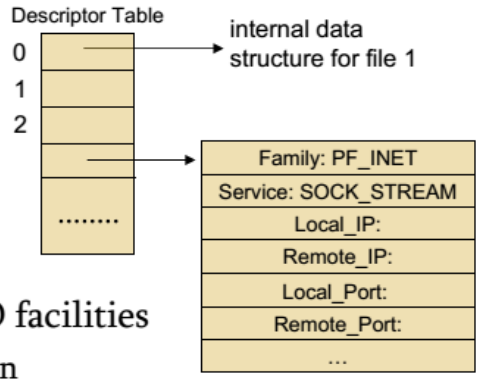
Berkley Sockets

- Universally known as **Sockets**
- It is an abstraction through which an application may send and receive data
- Provide **generic access** to interprocess communication services
  - e.g. IPX/SPX, Appletalk, TCP/IP
- Standard API for networking

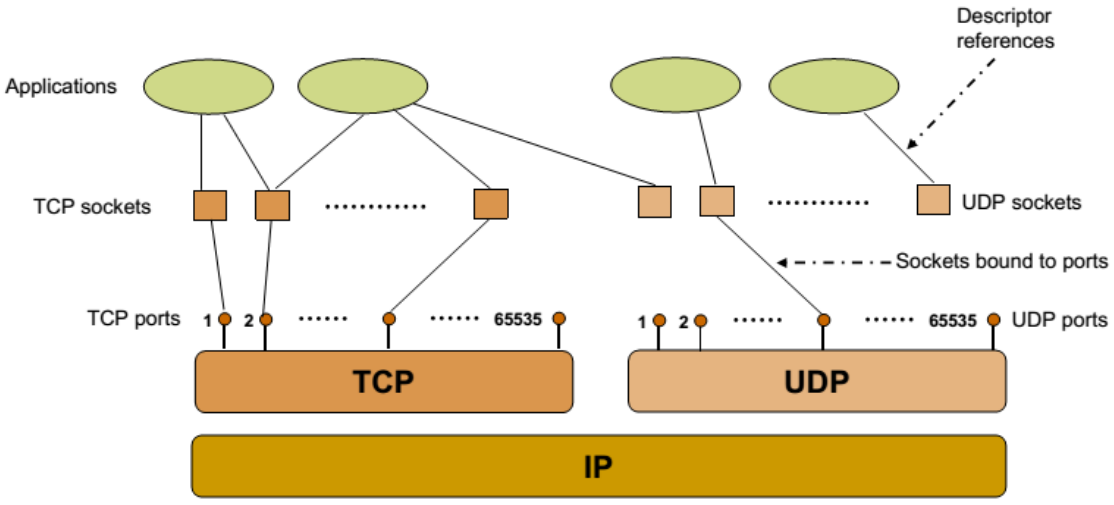


Sockets

- Uniquely identified by
  - an internet address
  - an end-to-end protocol (e.g. TCP or UDP)
  - a port number
- Two types of (TCP/IP) sockets
  - Stream sockets (e.g. uses TCP)
    - provide reliable byte-stream service
  - Datagram sockets (e.g. uses UDP)
    - provide best-effort datagram service
    - messages up to 65.500 bytes
- Socket extend the convectional UNIX I/O facilities
  - file descriptors for network communication
  - extended the read and write system calls



Sockets



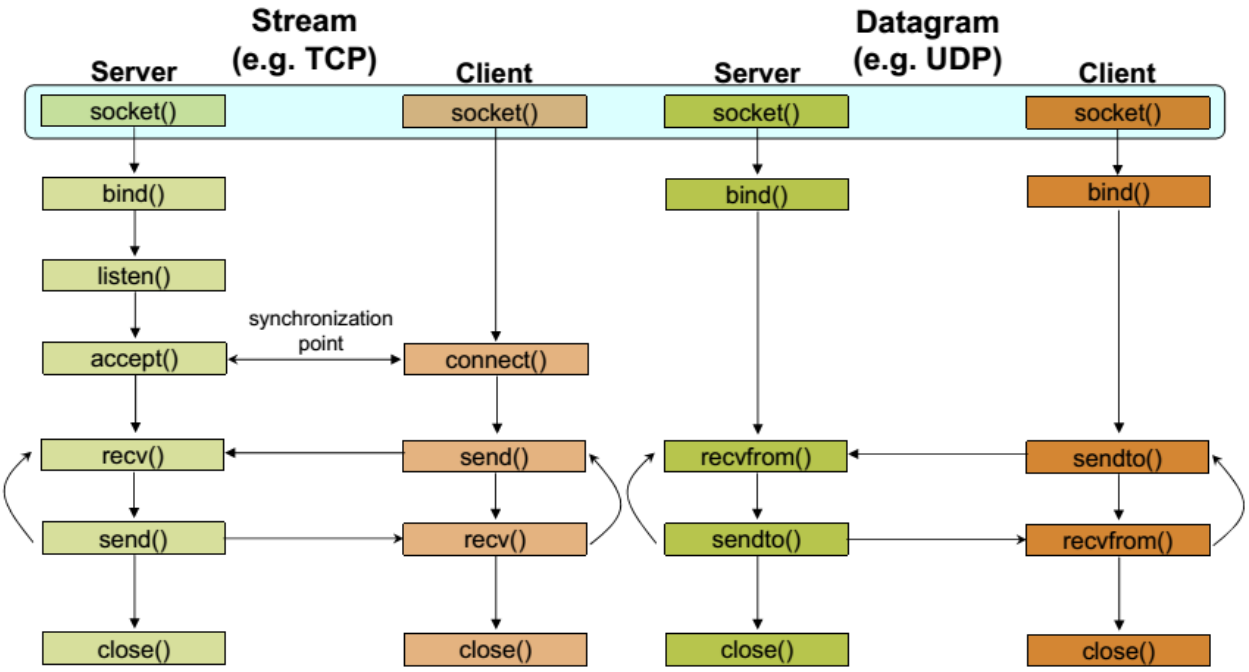
Client-Server communication

- **Server**
  - passively waits for and responds to clients
  - **passive** socket
- **Client**
  - initiates the communication
  - must know the address and the port of the server
  - **active** socket

Sockets - Procedures

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Client - Server Communication - Unix

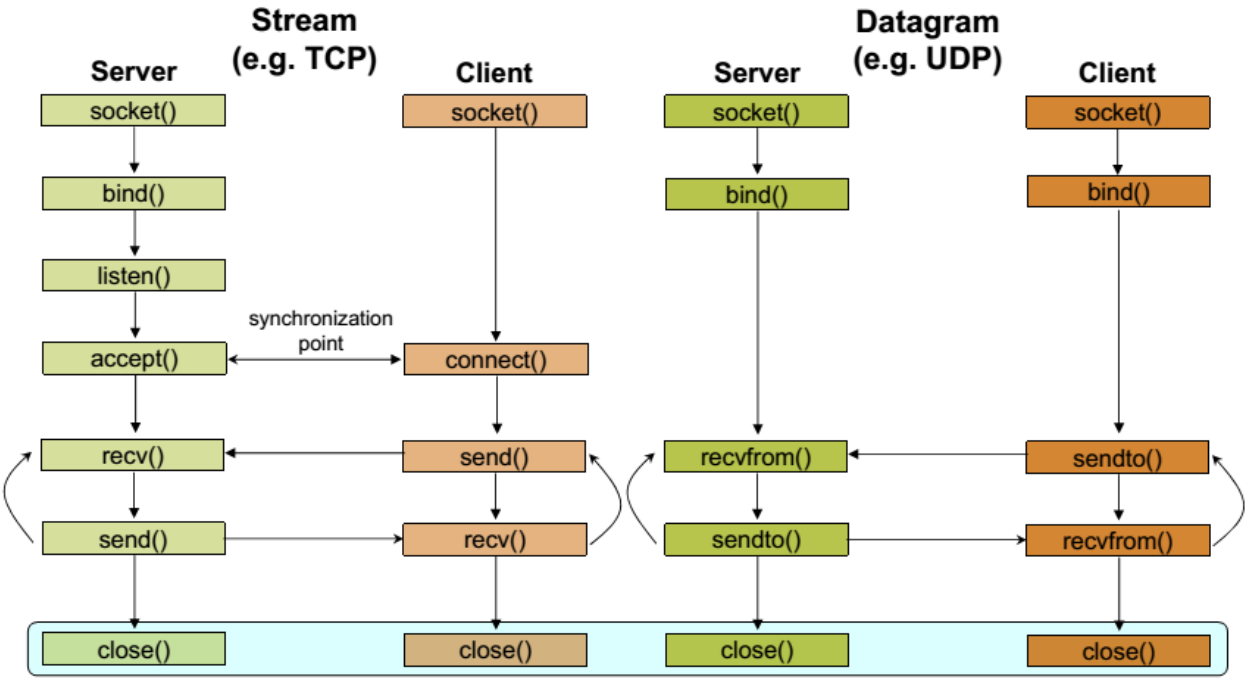


Socket creation in C: `socket()`

- `int sockid = socket(family, type, protocol);`
  - **sockid**: socket descriptor, an integer (like a file-handle)
  - **family**: integer, communication domain, e.g.,
    - PF_INET, IPv4 protocols, Internet addresses (typically used)
    - PF_UNIX, Local communication, File addresses
  - **type**: communication type
    - SOCK_STREAM - reliable, 2-way, connection-based service
    - SOCK_DGRAM - unreliable, connectionless, messages of maximum length
  - **protocol**: specifies protocol
    - IPPROTO_TCP IPPROTO_UDP
    - usually set to 0 (i.e., use default protocol)
  - upon failure returns -1

☞ NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

Client - Server Communication - Unix



Socket close in C: `close()`

- When finished using a socket, the socket should be closed
- `status = close(sockid);`
  - **sockid**: the file descriptor (socket being closed)
  - **status**: 0 if successful, -1 if error
- Closing a socket
  - closes a connection (for stream socket)
  - frees up the port used by the socket

Specifying Addresses

- Socket API defines a **generic** data type for addresses:

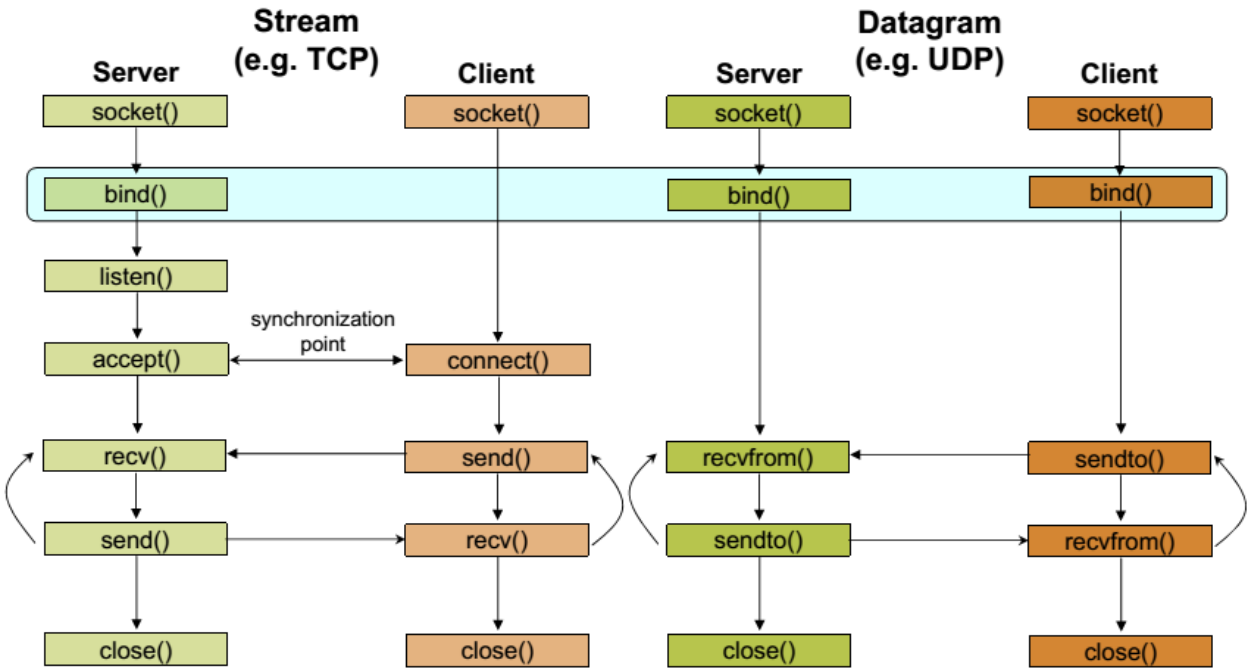
```
struct sockaddr {
   unsigned short sa_family;  /* Address family (e.g. AF_INET) */
   char sa_data[14];          /* Family-specific address information */
}
```

- Particular form of the sockaddr used for **TCP/IP** addresses:

```
struct in_addr {
   unsigned long s_addr;          /* Internet address (32 bits) */
}
struct sockaddr_in {
   unsigned short sin_family;     /* Internet protocol (AF_INET) */
   unsigned short sin_port;       /* Address port (16 bits) */
   struct in_addr sin_addr;     /* Internet address (32 bits) */
   char sin_zero[8];              /* Not used */
}
```

☞ **Important**: sockaddr_in can be casted to a sockaddr

Client - Server Communication - Unix



Assign address to socket: bind()

- associates and reserves a port for use by the socket
- `int status = bind(sockid, &addrport, size);`
  - **sockid**: integer, socket descriptor
  - **addrport**: struct sockaddr, the (IP) address and port of the machine
    - for TCP/IP server, internet address is usually set to INADDR_ANY, i.e., chooses any incoming interface
  - **size**: the size (in bytes) of the addrport structure
  - **status**: upon failure -1 is returned

bind() - Example with TCP

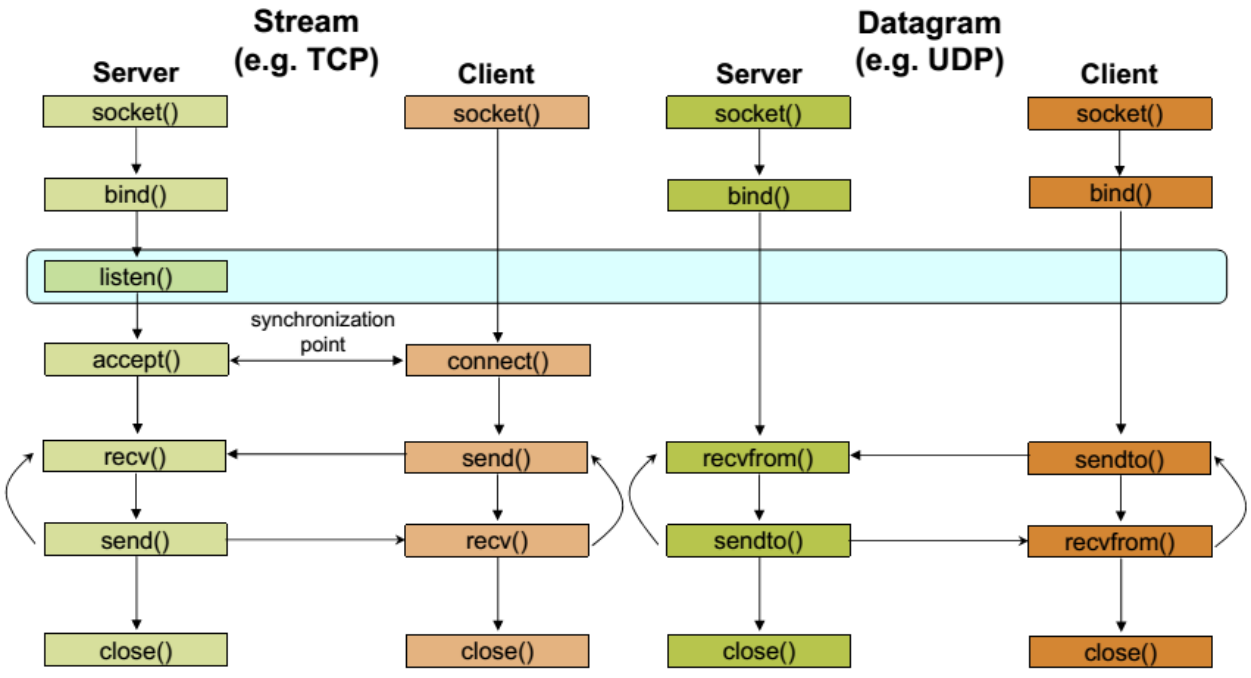
```
int sockid;
struct sockaddr_in addrport;
sockid = socket(PF_INET, SOCK_STREAM, 0);

addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockid, (struct sockaddr *) &addrport, sizeof(addrport))!= -1) {
   …}
```

Skipping the `bind()`

- bind can be skipped for both types of sockets
- Datagram socket:
  - if only sending, no need to bind. The OS finds a port each time the socket sends a packet
  - if receiving, need to bind
- Stream socket:
  - destination determined during connection setup
  - don't need to know port sending from (during connection setup, receiving end is informed of port)

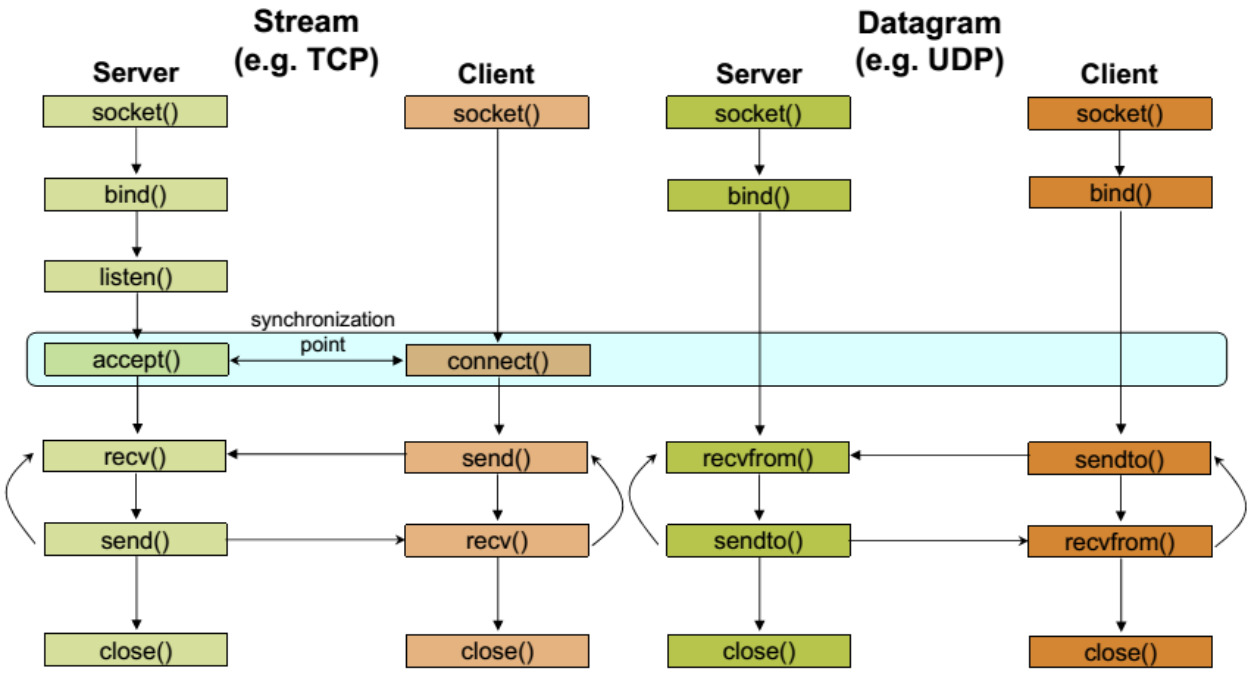
Client - Server Communication - Unix



Assign address to socket: `bind()`

- Instructs TCP protocol implementation to listen for connections
- `int status = listen(sockid, queueLimit);`
  - **sockid**: integer, socket descriptor
  - **queuelen**: integer, # of active participants that can "wait" for a connection
  - **status**: 0 if listening, -1 if error
- `listen()` is **non-blocking**: returns immediately
- The listening socket (sockid)
  - is never used for sending and receiving
  - is used by the server only as a way to get new sockets

Client - Server Communication - Unix



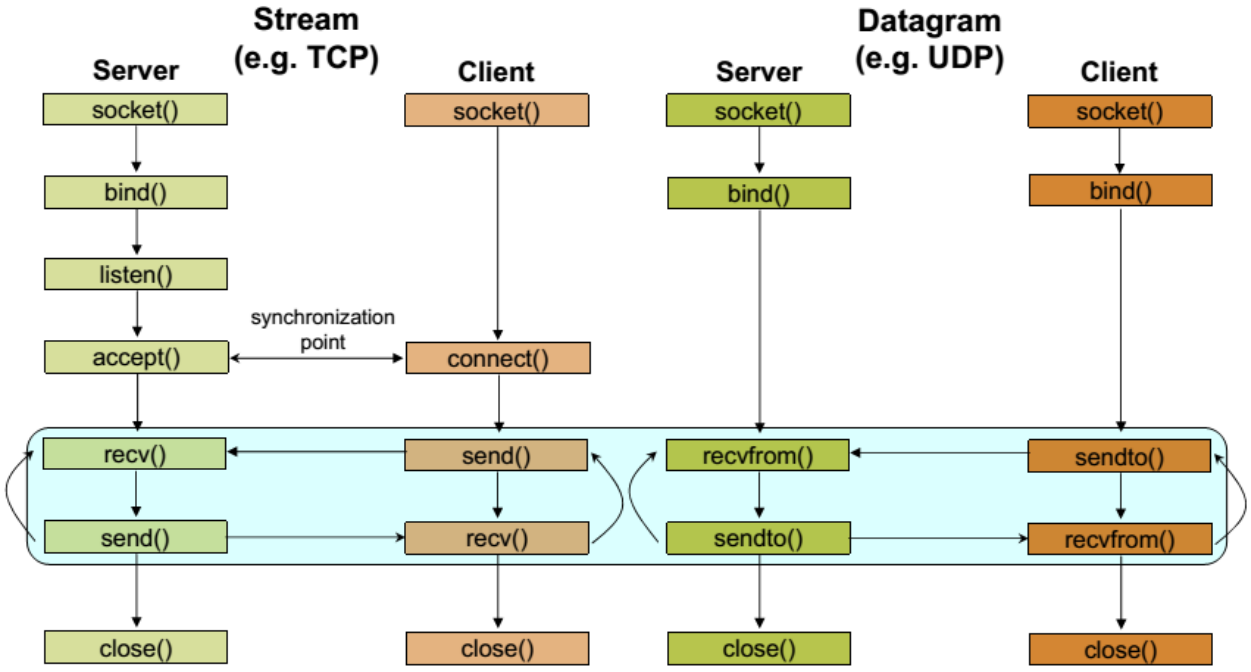
Establish Connection: `connect()`

- The client establishes a connection with the server by calling `connect()`
- `int status = connect(sockid, &foreignAddr, addrlen);`
  - **sockid**: integer, socket to be used in connection
  - **foreignAddr**: struct sockaddr: address of the passive participant
  - **addrlen**: integer, sizeof(name)
  - status: 0 if successful connect, -1 otherwise
- `connect()` is **blocking**

Incoming Connection: `accept()`

- The server gets a socket for an incoming client connection by calling `accept()`
- `int s = accept(sockid, &clientAddr, &addrLen);`
  - **s**: integer, the new socket (used for data-transfer)
  - **sockid**: integer, the orig. socket (being listened on)
  - **clientAddr**: struct sockaddr, address of the active participant
    - filled in upon return
  - **addrLen**: sizeof(clientAddr): value/result parameter
    - must be set appropriately before call
    - adjusted upon return
- `accept()`
  - is **blocking**: waits for connection before returning
  - dequeues the next connection on the queue for socket (sockid)

Client - Server Communication - Unix




```
 GNU nano 2.2.6                                          File: echo.c

#include <stdio.h> /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), connect(), recv() and send() */
#include <arpa/inet.h> /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h> /* for atoi() and exit() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for close() */
#define MAXPENDING 5 /* Maximum outstanding connection requests */
void DieWithError(char *errorMessage); /* Error handling function */
void HandleTCPClient(int clntSocket); /* TCP client handling function */
```

```c
int main(int argc, char *argv[]) {
    int servSock; /* Socket descriptor for server */
    int clntSock; /* Socket descriptor for client */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned short echoServPort; /* Server port */
    unsigned int clntLen; /* Length of client address data structure */
    if (argc != 2) { /* Test for correct number of arguments */
        fprintf(stderr, "Usage: %s <Server Port>\n", argv[0]);
        exit(1);    }
    echoServPort =atoi(argv[1]); /* First arg: local port */
    /* Create socket for incoming connections */
    if ((servSock =socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");
        /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr)); /* Zero out structure */
    echoServAddr.sin_family = AF_INET; /* Internet address family */
    echoServAddr.sin_addr.s_addr =htonl(INADDR_ANY); /* Any incoming interface */
    echoServAddr.sin_port =htons(echoServPort); /* Local port */
    /* Bind to the local address */
    if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");
    /* Mark the socket so it will listen for incoming connections */
    if (listen(servSock, MAXPENDING) < 0)  DieWithError("listen() failed");
    for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        clntLen = sizeof(echoClntAddr);
        /* Wait for a client to connect */
        if ((clntSock=accept(servSock, (struct sockaddr *) &echoClntAddr,&clntLen)) < 0)
            DieWithError("accept() failed");
        /* clntSock is connected to a client! */
        printf("Handling client %s\n",inet_ntoa(echoClntAddr.sin_addr));
        HandleTCPClient(clntSock);
    }
}
```

```c
#define RCVBUFSIZE 32 /* Size of receive buffer */
void HandleTCPClient(int clntSocket)
{
    char echoBuffer[RCVBUFSIZE]; /* Buffer for echo string */
    int recvMsgSize; /* Size of received message */
    /* Receive message from client */
    if ((recvMsgSize =recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
        DieWithError("recv() failed");
    /* Send received string and receive again until end of transmission */
    while (recvMsgSize > 0) /* zero indicates end of transmission */
    {
    /* Echo message back to client */
        if (send(clntSocket, echoBuffer, recvMsgSize, 0) != recvMsgSize)
            DieWithError("send() failed");
        /* See if there is more data to receive */
        if ((recvMsgSize =recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
            DieWithError("recv() failed");
    }
    close(clntSocket); /* Close client socket */
}
void DieWithError(char *errorMessage){
        exit(1);
}
```

Other useful functions

- bzero(char* c, int n): 0's n bytes starting at c
- gethostname(char *name, int len): gets the name of the current host
- gethostbyaddr(char *addr, int len, int type): converts IP hostname to structure containing long integer
- inet_addr(const char *cp): converts dotted-decimal char-string to long integer
- inet_ntoa(const struct in_addr in): converts long to dotted-decimal notation

- Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order

25

Exchanging data with stream socket

- `int count = send(sockid, msg, msgLen, flags);`
  - msg: const void[], message to be transmitted
  - msgLen: integer, length of message (in bytes) to transmit
  - flags: integer, special options, usually just 0
  - count: # bytes transmitted (-1 if error)
- `int count = recv(sockid, recvBuf, bufLen, flags);`
  - recvBuf: void[], stores received bytes
  - bufLen: # bytes received
  - flags: integer, special options, usually just 0
  - count: # bytes received (-1 if error)
- Calls are **blocking**
  - returns only after data is sent / received

Exchanging data with datagram socket

- `int count = sendto(sockid, msg, msgLen, flags, &foreignAddr, addrlen);`
  - msg, msgLen, flags, count: same with `send()`
  - **foreignAddr**: struct sockaddr, address of the destination
  - addrLen: sizeof(foreignAddr)
- `int count = recvfrom(sockid, recvBuf, bufLen, flags, &clientAddr, addrlen);`
  - recvBuf, bufLen, flags, count: same with `recv()`
  - **clientAddr**: struct sockaddr, address of the client
  - addrLen: sizeof(clientAddr)
- Calls are **blocking**
  - returns only after data is sent / received

Socket Options

- `getsockopt` and `setsockopt` allow socket options values to be queried and set, respectively
- **`int getsockopt (sockid, level, optName, optVal, optLen);`**
  - **sockid**: integer, socket descriptor
  - **level**: integer, the layers of the protocol stack (socket, TCP, IP)
  - **optName**: integer, option
  - **optVal**: pointer to a buffer; upon return it contains the value of the specified option
  - optLen: integer, in-out parameter

  it returns -1 if an error occured
- **`int setsockopt (sockid, level, optName, optVal, optLen);`**
  - optLen is now only an input parameter

Socket Options
- Table

optName	Type	Values	Description
SOL_SOCKET Level			
SO_BROADCAST	**int**	0,1	Broadcast allowed
SO_KEEPALIVE	**int**	0,1	Keepalive messages enabled (if implemented by the protocol)
SO_LINGER	**linger**{}	time	Time to delay close() return waiting for confirmation (see Section 6.4.2)
SO_RCVBUF	**int**	bytes	Bytes in the socket receive buffer (see code on page 44 and Section 6.1)
SO_RCVLOWAT	**int**	bytes	Minimum number of available bytes that will cause recv() to return
SO_REUSEADDR	**int**	0,1	Binding allowed (under certain conditions) to an address or port already in use (see Section 6.4 and 6.5)
SO_SNDLOWAT	**int**	bytes	Minimum bytes to send a packet
SO_SNDBUF	**int**	bytes	Bytes in the socket send buffer (see Section 6.1)
IPPROTO_TCP Level			
TCP_MAX	**int**	seconds	Seconds between keepalive messages.
TCP_NODELAY	**int**	0,1	Disallow delay for data merging (Nagle's algorithm)
IPPROTO_IP Level			
IP_TTL	**int**	0–255	Time-to-live for unicast IP packets
IP_MULTICAST_TTL	**unsigned char**	0–255	Time-to-live for multicast IP packets (see MulticastSender.c on page 81)
IP_MULTICAST_LOOP	**int**	0,1	Enables multicast socket to receive packets it sent
IP_ADD_MEMBERSHIP	**ip_mreq**{}	group address	Enables reception of packets addressed to the specified multicast group (see MulticastReceiver.c on page 83)—set only
IP_DROP_MEMBERSHIP	**ip_mreq**{}	group address	Disables reception of packets addressed to the specified multicast group—set only

Socket Options - Example

- Fetch and then double the current number of bytes in the socket's receive buffer

```
int rcvBufferSize;
int sockOptSize;
...
/* Retrieve and print the default buffer size */
sockOptSize = sizeof(recvBuffSize);
if (getsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcvBufferSize, &sockOptSize) < 0)
   DieWithError("getsockopt() failed");
printf("Initial Receive Buffer Size: %d\n", rcvBufferSize);

/* Double the buffer size */
recvBufferSize *= 2;

/* Set the buffer size to new value */
if (setsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcvBufferSize,
               sizeof(rcvBufferSize)) < 0)
 DieWithError("getsockopt() failed");
```

Dealing with blocking calls

- Many of the functions we saw block (by default) until a certain event
  - **accept**: until a connection comes in
  - **connect**: until the connection is established
  - **recv**, **recvfrom**: until a packet (of data) is received
    - what if a packet is lost (in datagram socket)?
  - **send**: until data are pushed into socket's buffer
  - **sendto**: until data are given to the network subsystem
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

Dealing with blocking calls

- Non-blocking Sockets
- Asynchronous I/O
- Timeouts

Non-blocking Sockets

- If an operation can be completed immediately, success is returned; otherwise, a failure is returned (usually -1)
  - `errno` is properly set, to distinguish this (blocking) failure from other - (`EINPROGRESS` for `connect`, `EWOULDBLOCK` for the other)
- 1st Solution: **`int fcntl (sockid, command, argument);`**
  - **sockid**: integer, socket descriptor
  - **command**: integer, the operation to be performed (**`F_GETFL`**, **`F_SETFL`**)
  - **argument**: long, e.g. **`O_NONBLOCK`**

  ☞ **`fcntl (sockid, F_SETFL, O_NONBLOCK);`**
- 2nd Solution: flags parameter of `send`, `recv`, `sendto`, `recvfrom`
  - **`MSG_DONTWAIT`**
  - not supported by all implementations

Signals

- Provide a mechanism for operating system to notify processes that certain events occur
  - e.g., the user typed the "interrupt" character, or a timer expired
- signals are delivered **asynchronously**
- upon signal delivery to program
  - it may be **ignored**, the process is never aware of it
  - the program is **forcefully terminated** by the OS
  - a **signal-handling routine**, specified by the program, is executed
    - this happens in a different thread
  - the signal is **blocked**, until the program takes action to allow its delivery
    - each process (or thread) has a corresponding **mask**
- Each signal has a **default behavior**
  - e.g. SIGINT (i.e., Ctrl+C) causes termination
  - it can be changed using `sigaction()`
- Signals can be **nested** (i.e., while one is being handled another is delivered)

Signals

- `int sigaction(whichSignal, &newAction, &oldAction);`
  - **whichSignal**: integer
  - **newAction**: `struct sigaction`, defines the new behavior
  - **oldAction**: `struct sigaction`, if not NULL, then previous behavior is copied
  - it returns 0 on success, -1 otherwise

```
struct sigaction {
   void (*sa_handler)(int); /* Signal handler */
   sigset_t sa_mask;        /* Signals to be blocked during handler execution */
   int sa_flags;            /* Flags to modify default behavior */
};
```

  - `sa_handler` determines which of the first three possibilities occurs when signal is delivered, i.e., it is not masked
    - `SIG_IGN`, `SIG_DFL`, address of a function
  - `sa_mask` specifies the signals to be blocked while handling **whichSignal**
    - **whichSignal** is always blocked
    - it is implemented as a set of boolean flags

```
int sigemptyset (sigset_t *set); /* unset all the flags */
int sigfullset (sigset_t *set);  /* set all the flags */
int sigaddset(sigset_t *set, int whichSignal); /* set individual flag */
int sigdelset(sigset_t *set, int whichSignal); /* unset individual flag */
```

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void DieWithError(char *errorMessage);
void InterruptSignalHandler(int signalType);

int main (int argc, char *argv[]) {
   struct sigaction handler;                    /* Signal handler specification structure */
   handler.sa_handler = InterruptSignalHandler; /* Set handler function */
   if (sigfillset(&handler.sa_mask) < 0)        /* Create mask that masks all signals */
      DieWithError ("sigfillset() failed");
   handler.sa_flags = 0;
   if (sigaction(SIGINT, &handler, 0) < 0)      /* Set signal handling for interrupt signals */
      DieWithError ("sigaction() failed");
   for(;;) pause();                             /* Suspend program until signal received */
   exit(0);
}

void InterruptHandler (int signalType) {
   printf ("Interrupt received. Exiting program.\n);
   exit(1);
}
```

Useful Functions

- **`int atoi(const char *nptr);`**
  - converts the initial portion of the string pointed to by `nptr` to int
- **`int inet_aton(const char *cp, struct in_addr *inp);`**
  - converts the Internet host address `cp` from the IPv4 numbers-and-dots notation into binary form (in network byte order)
  - stores it in the structure that `inp` points to.
  - it returns nonzero if the address is valid, and 0 if not
- **`char *inet_ntoa(struct in_addr in);`**
  - converts the Internet host address `in`, given in network byte order, to a string in IPv4 dotted-decimal notation

```
typedef uint32_t in_addr_t;

struct in_addr {
   in_addr_t s_addr;
};
```

Useful Functions

- **`int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`**
  - returns the address (IP and port) of the peer connected to the socket `sockfd`, in the buffer pointed to by `addr`
  - 0 is returned on success; -1 otherwise

- **`int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`**
  - returns the current address to which the socket `sockfd` is bound, in the buffer pointed to by `addr`
  - 0 is returned on success; -1 otherwise

Domain Name Service

- **`struct hostent *gethostbyname(const char *name);`**
  - returns a structure of type `hostent` for the given host name
  - `name` is a hostname, or an IPv4 address in standard dot notation

  e.g. **`gethostbyname("www.csd.uoc.gr");`**

- **`struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);`**
  - returns a structure of type hostent for the given host address addr of length len and address type `type`

```
struct hostent {
   char  *h_name;        /* official name of host */
   char **h_aliases;     /* alias list (strings) */
   int    h_addrtype;    /* host address type (AF_INET) */
   int    h_length;      /* length of address */
   char **h_addr_list;   /* list of addresses (binary in network byte order) */
}
#define h_addr h_addr_list[0]  /* for backward compatibility */
```

Domain Name Service

- **`struct servent *getservbyname(const char *name, const char *proto);`**
  - returns a `servent` structure for the entry from the database that matches the service name using protocol `proto`.
  - if `proto` is NULL, any protocol will be matched.

  e.g. **`getservbyname("echo", "tcp");`**

- **`struct servent *getservbyport(int port, const char *proto);`**
  - returns a `servent` structure for the entry from the database that matches the service name using port `port`

```
struct servent {
   char  *s_name;        /* official service name */
   char **s_aliases;     /* list of alternate names (strings)*/
   int    s_port;        /* service port number */
   char  *s_proto;       /* protocol to use ("tcp" or "udp")*/
}
```

مراحل ساخت یک proxy server ساده:

You will design a simple proxy server. Your server should expect to receive an HTTP request. It should then transmit a response back to the location from where the request came. The response should include some information of the request, as well as some indication that it passed through the proxy. For example, one obvious (and easy) solution would be to return a text message saying, "The proxy received the following text: XXX", where XXX is the text of the message received. If this is done correctly, your web browser will display the raw text received. If you want to do something more creative (e.g., if you know some more about HTTP and actually want to return a web page of some sort), please do so.

Your proxy needs to perform the following steps:

1. Listen on a given port (the one you told your web browser to forward requests to) for a request. HTTP requests are issued using the TCP protocol (i.e., the request expects to reach a socket with its protocol option set to SOCK_STREAM). This means that communication is connection-oriented.
   (a) create a socket of type SOCK_STREAM
   (b) bind the socket to the port that you configured the web browser to forward its requests to
   (c) listen for an incoming request
2. Accept an incoming connection when it arrives onto a socket (recall from class that the connection is established on a new socket).
3. Receive the data being transmitted off the socket.
4. Compute a response
5. Send the response out on the same socket (we are taking advantage of the duplex capabilities of the socket and the bi-directional nature of the connection).
6. Close the socket that performed the data transfer. Either close the other socket as well, or have the program loop so that the proxy will accept other requests
7. Run your proxy and initiate a request in your browser (configured to use the proxy) for some web page.
8. When you are done with your assignment, don't forget to reconfigure your web browser so that it no longer goes through the proxy or you will have trouble accessing the web.

IP port aliasing

Linux networking IP port aliasing tutorial

This tutorial works through an example of how IP aliasing can be used to provide multiple network addresses on a single physical interface. This demonstrates using IP version 4 addresses only.

Most distributions include IP aliasing compiled into the kernel. If that is not the case for your particular distro then you can load the module using the insmod command.

One reason for using this could be to make a computer look as though it is multiple computers, so for example you could have one server that is acting as both a gateway (router) and a DHCP server and DNS using 3 different IP addresses, perhaps with a future plan to use a hardware router and to move the functionality to seperate DNS and DHCP servers. Or indeed the opposite you could decide to replace the 3 different hardware devices with a single server to reduce the administration overhead.

In this case you can have 3 different addresses which are all on the same computer without having to install lots of physical network interfaces.

Another reason, which is the one I will use in this tutorial, is that you want to have the computer on two different logical network subnets whilst using a single physical interface. The reason I originally needed to do

this is that I purchased a network device that comes pre-configured with an IP address on the 192.168.0.0/24 address range, but my linux machine is on the 192.168.1.0/24 address range. I therefore configured an alias so that I can connect to the new device to configure it onto my 192.168.1.0/24 network address.

Changing the address dynamically

The first step is to identify the port number of the current interface. This is done using the ifconfig file.

The relevant output in this case is:

```
eth1      Link encap:Ethernet  HWaddr 00:0d:61:0b:d9:a0

          inet addr:192.168.1.1  Bcast:192.168.1.255
Mask:255.255.255.0

          inet6 addr: fe80::20d:61ff:fe0b:d9a0/64 Scope:Link

          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1

          RX packets:213915 errors:0 dropped:0 overruns:0
frame:0

          TX packets:211302 errors:0 dropped:0 overruns:0
carrier:0

          collisions:0 txqueuelen:1000

          RX bytes:140569143 (134.0 MB)  TX bytes:188664521
(179.9 MB)

          Interrupt:21 Base address:0xa000
```

We are using eth1 at the moment, so we will add an alias as eth1:1.

To add an alias IP address run

```
sudo ifconfig eth1:0 192.168.0.1
```

This creates an alias on eth1 with ip address 192.168.0.1. This will take the default network mask unless it is specified using the netmask option.

Thew new alias can be viewed using the ifconfig command.

```
eth1:0    Link encap:Ethernet  HWaddr 00:0d:61:0b:d9:a0

          inet addr:192.168.0.1  Bcast:192.168.0.255
Mask:255.255.255.0

          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1

          Interrupt:21 Base address:0xa000
```

Note that the hardware address (MAC address) HWaddr is the same as this is still the same physical interface showing that this is the same interface.

There are no interface statistics for the alias interface as these are included in the physical interface. To get per address statistics then accounting rules would need to be used through the iptables command.

A network route is automatically added to the routing table for the subnet with the interface included, but any additional routes will need to be added manually.

```
Kernel IP routing table

Destination     Gateway         Genmask         Flags   MSS
Window  irtt Iface

192.168.1.0     0.0.0.0         255.255.255.0   U         0 0
0 eth1

192.168.0.0     0.0.0.0         255.255.255.0   U         0 0
0 eth1

...
```

It should then be possible to access devices on the 192.168.0.0 network.

You may need to make changes to any firewall rules to allow access to the new network range. In my case I am running shorewall, which by default just uses the physical interface for it's rules. So a restart of shorewall and it worked. If a separate security policy is required for the new network then that needs to be configured explicitly in the firewall rules.

Making the changes permanent

The commands run above will make the changes to a live system, but these will all be lost when the system reboots. The commands could be added to a script that is called during startup, but the correct solution is to add these to the appropriate network configuration files.

The following is for Ubuntu Linux. This may differ on different distros. If the /etc/network/interfaces file does not exist on your distro then you can just add the earlier commands into /etc/rc.d/rc.local or a similar startup file.

The port definition needs to go into the interfaces file as shown below:

```
iface eth1 inet static

address 192.168.1.1

netmask 255.255.255.0

auto eth1


iface eth1:0 inet static

address 192.168.0.1

netmask 255.255.255.0
```

```
auto eth1:0
```

The last 4 lines have been added for this interface. The other interfaces have been excluded from the screen-capture above.

توضیحات لازم برای برنامه ی چت:

Project #1A – Chat Client

Section I: Introduction

The chat application allows multiple chat clients to connect to a chat server. Connected clients can: list the members already logged in, log in with a username, exchange messages with other logged in users, and log out. The server has to accept and maintain connections to all the clients and relay chat messages between them.

Section II: Chat Client Overview

The chat client application has to do the following tasks:

1. **Connect** to a chat server. The client obtains the server IP address and the port it is listening on from command-line arguments. For example:

   >./client 128.32.48.187 10000

   Then, the client immediately attempts to connect to the server socket using TCP. If it fails, the client application prints an error message to the console and returns. To print the error message, the client must call function *perror* (defined in stdio.h) with the name of the socket function that failed in lowercase. For example:

perror("socket");

If, on the other hand, the connection succeeds, the client continues to step 2.

2. **Accept console text commands** from the user. The client must be able to handle the following commands. Parameters are enclosed in <>:
    a. **login <username>**
    b. **list**
    c. **sendto <username> <message>**
    d. **logout**
    e. **exit**

    Each line must begin with a command. The parameters (if any) are separated by a space. The commands and parameters are described in more detail below. In order to implement these commands correctly, the client will have to send and receive the following messages to/from the server:
    a. **Send messages** to the server. The client must be able to send the following messages to the server. These messages are described in detail in section 4 of the document:
        i. **Login Message**
        ii. **List Message**
        iii. **Sendto Message**
        iv. **Logout Message**
    b. **Accept incoming messages** from the server. The client must be able to properly react to the following messages. These messages are shown in section 5 of the document:
        i. **List Message**
        ii. **Response Message**
        iii. **Sendto Message**

Section III: Console Commands

We now present the console commands that a client must handle. *Note: whenever a client prints an error message, it should not perform any other action.*

- **login**: sends a login message to register with the server using the provided username.
    - **Parameter(s):** a string to be used as the username.
    - **Rules:**
        - The string may not contain any white space. Screen names are treated as case sensitive.
        - Any extra parameters after the username must cause the client to print an error. The error must also be printed if the username is more than 20 characters in length or if the username is not present. The following is the error:
          Bad username
    - **Examples:**
        - The following is an example of a valid command:
          login John

        - The following is an example of an invalid command:
          login John Doe

- **list:** sends a list message requesting the list of users already logged in.
    - **Parameter(s):** none
    - **Rules:**
        - Any extra parameters must cause the client to print the following error:
          Invalid command
    - **Examples:**
        - The following is an example of a valid command:
          list
        - The following is an example of an invalid command:
          list bob
- **sendto:** sends a sendto message to the server to communicate with another client.
    - **Parameters:** The user name of the destination client and the text to send.
    - **Rules:**
        - If either or both of the parameters are missing or if the username contains more than 20 characters, the following error message should be printed to the console:
          Bad username
        - If there are more than 65535 characters of text, the following error message should be printed to the console:
          Bad message
    - **Examples:**
        - The following is an example of a valid command:
          sendto John Hello, how are you?
        - The following is an example of an invalid command:
          sendto JohnHello
- **logout**: sends a logout message to the server.
    - **Parameters**: none.
    - **Rules:**
        - Any extra parameters must cause the client to print the error:
          Invalid command
        - The client should not close the connection with the server when executing this command. After executing this command, the client should still be able to execute the list and login commands.
    - **Examples:**
        - The following is an example of a valid command:
          logout
        - The following is an example of an invalid command:
          logout Joe
- **exit:** closes the client socket and exits the program.

- **Parameters:** Takes no parameters
- **Rules:**
  - Any extra parameters must cause the client to print the error: *Invalid command*
- **Examples**:
  - The following is an example of a valid command: *exit*
  - The following is an example of an invalid command *exit client*

Protocol Overview: For each message that the client sends to the server the server will send a message back. For a successful List Message the server will return a List Message back (see the messages specification for details). For all other messages or an unsuccessful List Message, the server will respond with a Response Message. The client never sends messages back in response to messages received from the server.

Section IV: Messages Sent from Client to Server

The following is the structure for messages the client sends to the server. In all of these diagrams, B is a shortcut for Byte and XB refers to a variable length field. *Also, the client must expect a response from the server whenever it sends a message and must block user input until it receives that message.*

- **Login message:**

  - **Field values:**
    - MSG TYPE: Set to 0x01.
    - UNAME: Login username.
  - **Rules:**
    - UNAME must be padded to twenty bytes (i.e. if Bob is the user name, it must be followed by 17 null bytes).
    - UNAME does not have to be null-terminated (i.e. can contain a user name with 20 characters).
    - UNAME cannot contain any white space.
- **List message:**

  - **Field values:**
    - MSG TYPE: Set to 0x02.
- **Sendto message:**

- **Field values:**
  - MSG TYPE: Set to 0x03.
  - UNAME: User name of client to whom we are sending the message (same rules as for the login messages apply).
  - T LEN: Number of bytes in the text field (this value is treated as unsigned).
  - TEXT: Message we are sending to the client. Does not have to be null terminated.

- **Logout message:**

  - **Field values:**
    - MSG TYPE: Set to 0x04

Section V: Messages Sent from Server to Client

The following is the structure for messages the client receives from the server. Again, in all of these diagrams, B is a shortcut for Byte and XB refers to a variable length field.

- **List message:** sent in response to successful list messages received from the client.

  - **Field values:**
    - MSG TYPE: Set to 0x02.
    - UN NUM: Number of user names in this message (this value is treated as unsigned).
    - UNAMES: The user names of the clients that are logged in. Each user name is padded to 20 bytes as specified earlier in the document.
  - **Rules:**
    - The list should be printed one user name after another, ending with the text "<end list>". For example:
      Bob
      Joe
      <end list>

- **Response message:** sent in response to all messages received from the client, except where list messages are appropriate.

  - **Field values:**
    - MSG TYPE: Set to 0x05
    - RES TYPE: Set to 0x01 for OK, 0x02 for malformed packet, 0x03 for failure. OK means the command was successful. An example of a malformed packet is one that does not have a valid MSG TYPE. An example of failure is when a client tries to log in with a taken username.
  - **Rules:**

- If the server finds that a list message from the client has a problem (i.e. the server cannot provide the list of logged in users), it will send a response message, NOT a list message.
- If result is ok, the client must print the following message:
  Success
- If result is failure, the client must print the following message:
  Failure
- If the result is malformed packet, there is something wrong with your code.

- **Sendto message:** sent whenever another client wishes to communicate with us.

MSG TYPE (1B)	UNAME (20B)	T LEN (2B)	TEXT (XB)

  - **Field values:**
    - MSG TYPE: Set to 0x03.
    - UNAME: User name of client who sent us the message (same rules as for the login messages apply).
    - T LEN: Number of bytes in the text field (this value is treated as unsigned).
    - TEXT: Message sent by the client. Does not have to be null terminated.
  - **Rules:**
    - Client should be able to receive this message whenever it is logged in.
    - Cannot assume that the values in the UNAME or the TEXT fields are null-terminated.
    - Whenever this message is received, the client application has to print the message in the form “<username>:<text message>” to the console. For example if Bob sends “Hi how are you?” to John, then John’s client should print out:
      Bob:Hi How are you?

Project #1B – Chat Server

Section I: Introduction

The chat server is responsible for providing a mechanism for the clients to communicate. This includes allowing clients to login so they can be identified by username and forwarding messages from one client destined to another. This also includes listing the usernames of the clients that are currently logged in to facilitate communication.

Section II: Chat Server Overview

The chat server has to do the following tasks:

1. **Listen** for incoming client connections at a specified port. The server obtains the port number from a command-line argument. For example:

   >./server 10000

   Then the server immediately attempts to create the socket using TCP and tries to listen for connections to the specified port number. If it fails, the server prints an error message to the console and returns. To print the error message, the server must call function *perror()* (defined in stdio.h) with the name of the socket function that failed in lowercase. For example:

   perror("socket")

   If on the other hand, the server is successful, it continues to step 2.
2. **Respond to messages** received by multiple clients. The server must be able to properly respond to the following messages:
   a. **Login Message**

b. **List Message**
c. **Sendto Message**
d. **Logout Message**

The structure of these messages and the actions the server must take when receiving them are provided in the following section. *Additionally, the server must be able to handle multiple clients simultaneously using select().*

Section III: Responding to Messages

The following is the structure of the messages the server receives from the client and the actions the server must take. B is a shortcut for Byte and XB refers to a variable length field. Assume big endian number representation for these messages.

- **Login message:**
  - **Structure of message**

    - **Field values:**
      - MSG TYPE: Set to 0x01
      - UNAME: Login username
  - **Actions**
    - If UNAME contains whitespace, if it starts with a null byte, or if it is already taken by another user, send failure response.
    - If the client who sent this message is already logged in, also send failure response.
    - Otherwise, associate the username with the connection and send OK response
  - **Structure of response**

    - **Field values:**
      - MSG TYPE: Set to 0x05
      - RES TYPE: Set to 0x01 for OK, 0x02 for Malformed, 0x03 for failure.
- **List Message:**
  - **Structure of message**

    - **Field values:**
      - MSG TYPE: Set to 0x02
  - **Actions**
    - Fill response message with username of every logged in client.

- **Structure of response**

MSG TYPE (1B)	UN NUM (2B)	UNAMES (XB)

  - **Field values**
    - MSG TYPE: Set to 0x02
    - UN NUM: Set to number of logged in clients (this value is treated as unsigned)
    - UNAMES: The user names of the clients that are logged in. Each user name is padded to 20 bytes as specified in the previous document.

- **Sendto Message:**
  - **Structure of message**

MSG TYPE (1B)	UNAME (20B)	T LEN (2B)	TEXT (XB)

  - **Field values:**
    - MSG TYPE: Set to 0x03.
    - UNAME: User name of client to whom to forward the message.
    - T LEN: Number of bytes in the text field (this value is treated as unsigned).
    - TEXT: Message sent by the client. Does not have to be null terminated.
  - **Actions**
    - *In the following explanation, this message was received from client A and the target of the message is client B.*
    - If client A is not logged in or cannot find client B (who is specified with UNAME), send failure response to client A.
    - Otherwise, replace the contents of UNAME with client A's UNAME and send the message to client B. If this fails, transmit a failure response to client A.
    - Otherwise send OK response to client A.
  - **Structure of response**

MSG TYPE (1B)	RES TYPE (1B)

  - **Field values:**
    - MSG TYPE: Set to 0x05
    - RES TYPE: Set to 0x01 for OK, 0x02 for Malformed, 0x03 for failure.

- **Logout Message:**
  - **Structure of message**

MSG TYPE (1B)

  - **Field values:**

- MSG TYPE: Set to 0x04

  - **Actions**
    - If client is not logged in, send failure response. Otherwise send OK response.
  - **Structure of response**

    - **Field values:**
      - MSG TYPE: Set to 0x05
      - RES TYPE: Set to 0x01 for OK, 0x02 for Malformed, 0x03 for failure.

- **All Other Messages:**
  - **Actions**
    - Send response indicating that we have received malformed packet.
    - Close connection with client.

Section IV: Other Requirements

- Your server should be robust to unexpected messages and should close the connection with the client that caused that message instead of crashing. Other clients should not be affected.
- There should not be any memory leaks
- The server should interact properly with the client binary that was been provided as part of the client test script.

Section V: Grade Breakdown

- 95% - Passing all test-cases that test the functionality of the server (including its ability to avoid crashes).
- 5% - Not having any memory leaks. Valgrind is a good tool for this.

مروری بر thread

Processes vs. Threads

Processes

- Multiple simultaneous programs
- Independent memory space
- Independent open file-descriptors

Threads

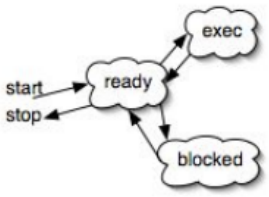
- Multiple simultaneous functions
- Shared memory space
- Shared open file-descriptors

Threads Examples

- Graphical User Interfaces (GUIs)
  - The GUI is usually put on a separate thread from the "app engine"
  - GUI remains responsive even if app blocks for processing
- Web Browser Tabs
  - Each tab is managed by a separate thread for rendering
  - Web pages render "simultaneously"
  - Note: Google Chrome actually uses a separate process per tab

Threads

- One copy of the heap
- One copy of the code
- Multiple stacks
- Life Cycle



pthread

- #include <pthread.h>
- Define a worker function
  - void *foo(void *args) { }
- Initialize pthread_attr_t
  - pthread_attr_t attr;
  - pthread_attr_init(attr);
- Create a thread
  - pthread_t thread;
  - pthread_create(&thread, &attr, worker_function, arg);
- Exit current thread
  - pthread_exit(status)

Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS     5

void *print_hello(void *threadid)
{
    long tid;
    tid = (long) threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for (t = 0; t < NUM_THREADS; t++)
    {
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(threads + t, NULL, print_hello, (void *) t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Thread Management

- pthread_join(threadid, status)
- pthread_detach(threadid)

Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS     5

void *print_hello(void *threadid) {
    long tid;
    tid = (long) threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(threads + t, NULL, print_hello, (void *) t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    /* wait for all threads to complete */
    for (t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }
    pthread_exit(NULL);
}
```

Compiling

- Use "-pthread"

Programming with pcap

Ok, let's begin by defining who this document is written for. Obviously, some basic knowledge of C is required, unless you only wish to know the basic theory. You do not need to be a code ninja; for the areas likely to be understood only by more experienced programmers, I'll be sure to describe concepts in greater detail. Additionally, some basic understanding of networking might help, given that this is a packet sniffer and all. All of the code examples presented here have been tested on FreeBSD 4.3 with a default kernel.

Getting Started: The format of a pcap application

The first thing to understand is the general layout of a pcap sniffer. The flow of code is as follows:

1. We begin by determining which interface we want to sniff on. In Linux this may be something like eth0, in BSD it may be xl1, etc. We can either define this device in a string, or we can ask pcap to provide us with the name of an interface that will do the job.
2. Initialize pcap. This is where we actually tell pcap what device we are sniffing on. We can, if we want to, sniff on multiple devices. How do we differentiate between them? Using file handles. Just like opening a file for reading or writing, we must name our sniffing "session" so we can tell it apart from other such sessions.
3. In the event that we only want to sniff specific traffic (e.g.: only TCP/IP packets, only packets going to port 23, etc) we must create a rule set, "compile" it, and apply it. This is a three phase process, all of which is closely related. The rule set is kept in a string, and is converted into a format that pcap can read (hence compiling it.) The compilation is actually just done by calling a function within our program; it does not involve the use of an external application. Then we tell pcap to apply it to whichever session we wish for it to filter.
4. Finally, we tell pcap to enter it's primary execution loop. In this state, pcap waits until it has received however many packets we want it to. Every time it gets a new packet in, it calls another function that we have already defined. The function that it calls can do anything we want; it can dissect the packet and print it to the user, it can save it in a file, or it can do nothing at all.
5. After our sniffing needs are satisfied, we close our session and are complete.

This is actually a very simple process. Five steps total, one of which is optional (step 3, in case you were wondering.) Let's take a look at each of the steps and how to implement them.

Setting the device

This is terribly simple. There are two techniques for setting the device that we wish to sniff on.

The first is that we can simply have the user tell us. Consider the following program:

```
#include <stdio.h>
#include <pcap.h>

int main(int argc, char *argv[])
{
         char *dev = argv[1];

         printf("Device: %s\n", dev);
         return(0);
}
```

The user specifies the device by passing the name of it as the first argument to the program. Now the string "dev" holds the name of the interface that we will sniff on in a format that pcap can understand (assuming, of course, the user gave us a real interface).

The other technique is equally simple. Look at this program:

```
#include <stdio.h>
#include <pcap.h>

int main(int argc, char *argv[])
{
        char *dev, errbuf[PCAP_ERRBUF_SIZE];

        dev = pcap_lookupdev(errbuf);
        if (dev == NULL) {
                fprintf(stderr, "Couldn't find default device: %s\n",
errbuf);
                return(2);
        }
        printf("Device: %s\n", dev);
        return(0);
}
```

In this case, pcap just sets the device on its own. "But wait, Tim," you say. "What is the deal with the errbuf string?" Most of the pcap commands allow us to pass them a string as an argument. The purpose of this string? In the event that the command fails, it will populate the string with a description of the error. In this case, if pcap_lookupdev() fails, it will store an error message in errbuf. Nifty, isn't it? And that's how we set our device.

Opening the device for sniffing

The task of creating a sniffing session is really quite simple. For this, we use pcap_open_live(). The prototype of this function (from the pcap man page) is as follows:

```
        pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int
to_ms,
            char *ebuf)
```

The first argument is the device that we specified in the previous section. snaplen is an integer which defines the maximum number of bytes to be captured by pcap. promisc, when set to true, brings the interface into promiscuous mode (however, even if it is set to false, it is possible under specific cases for the interface to be in promiscuous mode, anyway). to_ms is the read time out in milliseconds (a value of 0 means no time out; on at least some platforms, this means that you may wait until a sufficient number of packets arrive before seeing any packets, so you should use a non-zero timeout). Lastly, ebuf is a string we can store any error messages within (as we did above with errbuf). The function returns our session handler.

To demonstrate, consider this code snippet:

```
        #include <pcap.h>
        ...
        pcap_t *handle;

        handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
        if (handle == NULL) {
                fprintf(stderr, "Couldn't open device %s: %s\n", dev,
errbuf);
                return(2);
        }
```

This code fragment opens the device stored in the strong "dev", tells it to read however many bytes are specified in BUFSIZ (which is defined in pcap.h). We are

telling it to put the device into promiscuous mode, to sniff until an error occurs, and if there is an error, store it in the string errbuf; it uses that string to print an error message.

A note about promiscuous vs. non-promiscuous sniffing: The two techniques are very different in style. In standard, non-promiscuous sniffing, a host is sniffing only traffic that is directly related to it. Only traffic to, from, or routed through the host will be picked up by the sniffer. Promiscuous mode, on the other hand, sniffs all traffic on the wire. In a non-switched environment, this could be all network traffic. The obvious advantage to this is that it provides more packets for sniffing, which may or may not be helpful depending on the reason you are sniffing the network. However, there are regressions. Promiscuous mode sniffing is detectable; a host can test with strong reliability to determine if another host is doing promiscuous sniffing. Second, it only works in a non-switched environment (such as a hub, or a switch that is being ARP flooded). Third, on high traffic networks, the host can become quite taxed for system resources.

Not all devices provide the same type of link-layer headers in the packets you read. Ethernet devices, and some non-Ethernet devices, might provide Ethernet headers, but other device types, such as loopback devices in BSD and OS X, PPP interfaces, and Wi-Fi interfaces when capturing in monitor mode, don't.

You need to determine the type of link-layer headers the device provides, and use that type when processing the packet contents. The pcap_datalink() routine returns a value indicating the type of link-layer headers; see the list of link-layer header type values. The values it returns are the `DLT_` values in that list.

If your program doesn't support the link-layer header type provided by the device, it has to give up; this would be done with code such as

```
        if (pcap_datalink(handle) != DLT_EN10MB) {
                fprintf(stderr, "Device %s doesn't provide Ethernet headers -
not supported\n", dev);
                return(2);
        }
```

which fails if the device doesn't supply Ethernet headers. This would be appropriate for the code below, as it assumes Ethernet headers.

Filtering traffic

Often times our sniffer may only be interested in specific traffic. For instance, there may be times when all we want is to sniff on port 23 (telnet) in search of passwords. Or perhaps we want to highjack a file being sent over port 21 (FTP). Maybe we only want DNS traffic (port 53 UDP). Whatever the case, rarely do we just want to blindly sniff *all* network traffic. Enter pcap_compile() and pcap_setfilter().

The process is quite simple. After we have already called pcap_open_live() and have a working sniffing session, we can apply our filter. Why not just use our own if/else if statements? Two reasons. First, pcap's filter is far more efficient, because it does it directly with the BPF filter; we eliminate numerous steps by having the BPF driver do it directly. Second, this is a *lot* easier :)

Before applying our filter, we must "compile" it. The filter expression is kept in a regular string (char array). The syntax is documented quite well in the man page for tcpdump; I leave you to read it on your own. However, we will use simple test expressions, so perhaps you are sharp enough to figure it out from my examples.

To compile the program we call pcap_compile(). The prototype defines it as:

```
        int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int
optimize,
            bpf_u_int32 netmask)
```

The first argument is our session handle (pcap_t *handle in our previous example). Following that is a reference to the place we will store the compiled version of our filter. Then comes the expression itself, in regular string format. Next is an integer that decides if the expression should be "optimized" or not (0 is false, 1 is true. Standard stuff.) Finally, we must specify the network mask of the network the filter applies to. The function returns -1 on failure; all other values imply success.

After the expression has been compiled, it is time to apply it. Enter pcap_setfilter(). Following our format of explaining pcap, we shall look at the pcap_setfilter() prototype:

```
        int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

This is very straightforward. The first argument is our session handler, the second is a reference to the compiled version of the expression (presumably the same variable as the second argument to pcap_compile()).

Perhaps another code sample would help to better understand:

```
        #include <pcap.h>
        ...
        pcap_t *handle;              /* Session handle */
        char dev[] = "rl0";          /* Device to sniff on */
        char errbuf[PCAP_ERRBUF_SIZE];       /* Error string */
        struct bpf_program fp;               /* The compiled filter
expression */
        char filter_exp[] = "port 23";       /* The filter expression */
        bpf_u_int32 mask;            /* The netmask of our sniffing device
*/
        bpf_u_int32 net;             /* The IP of our sniffing device */

        if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
                fprintf(stderr, "Can't get netmask for device %s\n", dev);
                net = 0;
                mask = 0;
        }
        handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
        if (handle == NULL) {
                fprintf(stderr, "Couldn't open device %s: %s\n", dev,
errbuf);
                return(2);
        }
        if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
                fprintf(stderr, "Couldn't parse filter %s: %s\n", filter_exp,
pcap_geterr(handle));
                return(2);
        }
        if (pcap_setfilter(handle, &fp) == -1) {
                fprintf(stderr, "Couldn't install filter %s: %s\n",
filter_exp, pcap_geterr(handle));
                return(2);
        }
```

This program preps the sniffer to sniff all traffic coming from or going to port 23, in promiscuous mode, on the device rl0.

You may notice that the previous example contains a function that we have not yet discussed. pcap_lookupnet() is a function that, given the name of a device, returns one of its IPv4 network numbers and corresponding network mask (the network number is the IPv4 address ANDed with the network mask, so it contains only the network part of the address). This was essential because we needed to know the network mask in order to apply the filter. This function is described in the Miscellaneous section at the end of the document.

It has been my experience that this filter does not work across all operating systems. In my test environment, I found that OpenBSD 2.9 with a default kernel does support this type of filter, but FreeBSD 4.3 with a default kernel does not. Your mileage may vary.

The actual sniffing

At this point we have learned how to define a device, prepare it for sniffing, and apply filters about what we should and should not sniff for. Now it is time to actually capture some packets.

There are two main techniques for capturing packets. We can either capture a single packet at a time, or we can enter a loop that waits for *n* number of packets to be sniffed before being done. We will begin by looking at how to capture a single packet, then look at methods of using loops. For this we use pcap_next().

The prototype for pcap_next() is fairly simple:

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

The first argument is our session handler. The second argument is a pointer to a structure that holds general information about the packet, specifically the time in which it was sniffed, the length of this packet, and the length of his specific portion (incase it is fragmented, for example.) pcap_next() returns a u_char pointer to the packet that is described by this structure. We'll discuss the technique for actually reading the packet itself later.

Here is a simple demonstration of using pcap_next() to sniff a packet.

```
#include <pcap.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
       pcap_t *handle;                /* Session handle */
       char *dev;                     /* The device to sniff on */
       char errbuf[PCAP_ERRBUF_SIZE]; /* Error string */
       struct bpf_program fp;         /* The compiled filter */
       char filter_exp[] = "port 23"; /* The filter expression */
       bpf_u_int32 mask;              /* Our netmask */
       bpf_u_int32 net;               /* Our IP */
```

```
                struct pcap_pkthdr header;     /* The header that pcap gives us
*/
                const u_char *packet;          /* The actual packet */

                /* Define the device */
                dev = pcap_lookupdev(errbuf);
                if (dev == NULL) {
                        fprintf(stderr, "Couldn't find default device: %s\n",
errbuf);
                        return(2);
                }
                /* Find the properties for the device */
                if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
                        fprintf(stderr, "Couldn't get netmask for device %s:
%s\n", dev, errbuf);
                        net = 0;
                        mask = 0;
                }
                /* Open the session in promiscuous mode */
                handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
                if (handle == NULL) {
                        fprintf(stderr, "Couldn't open device %s: %s\n", dev,
errbuf);
                        return(2);
                }
                /* Compile and apply the filter */
                if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
                        fprintf(stderr, "Couldn't parse filter %s: %s\n",
filter_exp, pcap_geterr(handle));
                        return(2);
                }
                if (pcap_setfilter(handle, &fp) == -1) {
                        fprintf(stderr, "Couldn't install filter %s: %s\n",
filter_exp, pcap_geterr(handle));
                        return(2);
                }
                /* Grab a packet */
                packet = pcap_next(handle, &header);
                /* Print its length */
                printf("Jacked a packet with length of [%d]\n", header.len);
                /* And close the session */
                pcap_close(handle);
                return(0);
        }
```

This application sniffs on whatever device is returned by pcap_lookupdev() by putting it into promiscuous mode. It finds the first packet to come across port 23 (telnet) and tells the user the size of the packet (in bytes). Again, this program includes a new call, pcap_close(), which we will discuss later (although it really is quite self explanatory).

The other technique we can use is more complicated, and probably more useful. Few sniffers (if any) actually use pcap_next(). More often than not, they use pcap_loop() or pcap_dispatch() (which then themselves use pcap_loop()). To understand the use of these two functions, you must understand the idea of a callback function.

Callback functions are not anything new, and are very common in many API's. The concept behind a callback function is fairly simple. Suppose I have a program that is waiting for an event of some sort. For the purpose of this example, let's pretend that my program wants a user to press a key on the keyboard. Every time they press a key, I want to call a function which then will determine that to do. The function I am utilizing is a callback function. Every time the user presses a key, my program will call the callback function. Callbacks are used in pcap, but instead of being called when a user presses a key, they are called when pcap sniffs a packet. The two functions that one can use to define their callback is pcap_loop() and pcap_dispatch(). pcap_loop() and pcap_dispatch() are very similar in their usage of callbacks. Both of them call a callback function every time a packet is sniffed that meets our filter requirements (if any filter exists, of course. If not, then *all* packets that are sniffed are sent to the callback.)

The prototype for pcap_loop() is below:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

The first argument is our session handle. Following that is an integer that tells pcap_loop() how many packets it should sniff for before returning (a negative value means it should sniff until an error occurs). The third argument is the name of the callback function (just its identifier, no parentheses). The last argument is useful in some applications, but many times is simply set as NULL. Suppose we have arguments of our own that we wish to send to our callback function, in addition to the arguments that pcap_loop() sends. This is where we do it. Obviously, you must typecast to a u_char pointer to ensure the results make it there correctly; as we will see later, pcap makes use of some very interesting means of passing information in the form of a u_char pointer. After we show an example of how pcap does it, it should be obvious how to do it here. If not, consult your local C reference text, as an explanation of pointers is beyond the scope of this document. pcap_dispatch() is almost identical in usage. The only difference between pcap_dispatch() and pcap_loop() is that pcap_dispatch() will only process the first batch of packets that it receives from the system, while pcap_loop() will continue processing packets or batches of packets until

the count of packets runs out. For a more in depth discussion of their differences, see the pcap man page.

Before we can provide an example of using pcap_loop(), we must examine the format of our callback function. We cannot arbitrarily define our callback's prototype; otherwise, pcap_loop() would not know how to use the function. So we use this format as the prototype for our callback function:

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
    const u_char *packet);
```

Let's examine this in more detail. First, you'll notice that the function has a void return type. This is logical, because pcap_loop() wouldn't know how to handle a return value anyway. The first argument corresponds to the last argument of pcap_loop(). Whatever value is passed as the last argument to pcap_loop() is passed to the first argument of our callback function every time the function is called. The second argument is the pcap header, which contains information about when the packet was sniffed, how large it is, etc. The pcap_pkthdr structure is defined in pcap.h as:

```
struct pcap_pkthdr {
        struct timeval ts; /* time stamp */
        bpf_u_int32 caplen; /* length of portion present */
        bpf_u_int32 len; /* length this packet (off wire) */
};
```

These values should be fairly self explanatory. The last argument is the most interesting of them all, and the most confusing to the average novice pcap programmer. It is another pointer to a u_char, and it points to the first byte of a chunk of data containing the entire packet, as sniffed by pcap_loop().

But how do you make use of this variable (named "packet" in our prototype)? A packet contains many attributes, so as you can imagine, it is not really a string, but actually a collection of structures (for instance, a TCP/IP packet would have an Ethernet header, an IP header, a TCP header, and lastly, the packet's payload). This u_char pointer points to the serialized version of these structures. To make any use of it, we must do some interesting typecasting.

First, we must have the actual structures define before we can typecast to them. The following are the structure definitions that I use to describe a TCP/IP packet over Ethernet.

```
/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN 6

        /* Ethernet header */
        struct sniff_ethernet {
                u_char ether_dhost[ETHER_ADDR_LEN]; /* Destination host
address */
                u_char ether_shost[ETHER_ADDR_LEN]; /* Source host address */
                u_short ether_type; /* IP? ARP? RARP? etc */
        };

        /* IP header */
        struct sniff_ip {
                u_char ip_vhl;          /* version << 4 | header length >> 2 */
                u_char ip_tos;          /* type of service */
                u_short ip_len;         /* total length */
                u_short ip_id;          /* identification */
                u_short ip_off;         /* fragment offset field */
        #define IP_RF 0x8000            /* reserved fragment flag */
        #define IP_DF 0x4000            /* dont fragment flag */
        #define IP_MF 0x2000            /* more fragments flag */
        #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
                u_char ip_ttl;          /* time to live */
                u_char ip_p;            /* protocol */
                u_short ip_sum;         /* checksum */
                struct in_addr ip_src,ip_dst; /* source and dest address */
        };
        #define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
        #define IP_V(ip)                (((ip)->ip_vhl) >> 4)

        /* TCP header */
        typedef u_int tcp_seq;

        struct sniff_tcp {
                u_short th_sport;       /* source port */
                u_short th_dport;       /* destination port */
                tcp_seq th_seq;         /* sequence number */
                tcp_seq th_ack;         /* acknowledgement number */
                u_char th_offx2;        /* data offset, rsvd */
        #define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
                u_char th_flags;
        #define TH_FIN 0x01
        #define TH_SYN 0x02
        #define TH_RST 0x04
        #define TH_PUSH 0x08
        #define TH_ACK 0x10
        #define TH_URG 0x20
        #define TH_ECE 0x40
        #define TH_CWR 0x80
        #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
                u_short th_win;         /* window */
                u_short th_sum;         /* checksum */
```

```
                u_short th_urp;         /* urgent pointer */
};
```

So how does all of this relate to pcap and our mysterious u_char pointer? Well, those structures define the headers that appear in the data for the packet. So how can we break it apart? Be prepared to witness one of the most practical uses of pointers (for all of those new C programmers who insist that pointers are useless, I smite you).

Again, we're going to assume that we are dealing with a TCP/IP packet over Ethernet. This same technique applies to any packet; the only difference is the structure types that you actually use. So let's begin by defining the variables and compile-time definitions we will need to deconstruct the packet data.

```
/* ethernet headers are always exactly 14 bytes */
#define SIZE_ETHERNET 14

        const struct sniff_ethernet *ethernet; /* The ethernet header */
        const struct sniff_ip *ip; /* The IP header */
        const struct sniff_tcp *tcp; /* The TCP header */
        const char *payload; /* Packet payload */

        u_int size_ip;
        u_int size_tcp;
```

And now we do our magical typecasting:

```
        ethernet = (struct sniff_ethernet*)(packet);
        ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
        size_ip = IP_HL(ip)*4;
        if (size_ip < 20) {
                printf("   * Invalid IP header length: %u bytes\n", size_ip);
                return;
        }
        tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
        size_tcp = TH_OFF(tcp)*4;
        if (size_tcp < 20) {
                printf("   * Invalid TCP header length: %u bytes\n",
size_tcp);
                return;
        }
        payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
```

How does this work? Consider the layout of the packet data in memory. The u_char pointer is really just a variable containing an address in memory. That's what a pointer is; it points to a location in memory.

For the sake of simplicity, we'll say that the address this pointer is set to is the value X. Well, if our three structures are just sitting in line, the first of them (sniff_ethernet) being located in memory at the address X, then we can easily find the address of the structure after it; that address is X plus the length of the Ethernet header, which is 14, or SIZE_ETHERNET.

Similarly if we have the address of that header, the address of the structure after it is the address of that header plus the length of that header. The IP header, unlike the Ethernet header, does **not** have a fixed length; its length is given, as a count of 4-byte words, by the header length field of the IP header. As it's a count of 4-byte words, it must be multiplied by 4 to give the size in bytes. The minimum length of that header is 20 bytes.

The TCP header also has a variable length; its length is given, as a number of 4-byte words, by the "data offset" field of the TCP header, and its minimum length is also 20 bytes.

So let's make a chart:

Variable	Location (in bytes)
sniff_ethernet	X
sniff_ip	X + SIZE_ETHERNET
sniff_tcp	X + SIZE_ETHERNET + {IP header length}
payload	X + SIZE_ETHERNET + {IP header length} + {TCP header length}

The sniff_ethernet structure, being the first in line, is simply at location X. sniff_ip, who follows directly after sniff_ethernet, is at the location X, plus however much space the Ethernet header consumes (14 bytes, or SIZE_ETHERNET). sniff_tcp is after both sniff_ip and sniff_ethernet, so it is location at X plus the sizes of the Ethernet and IP headers (14 bytes, and 4 times the IP header length, respectively). Lastly, the payload (which doesn't have a single structure corresponding to it, as its contents depends on the protocol being used atop TCP) is located after all of them.

```
  GNU nano 2.2.6                File: pcap_test01.c

#include <stdio.h>
#include <pcap.h>

int main(int argc, char *argv[])
{
        char *dev, errbuf[PCAP_ERRBUF_SIZE];
                dev = pcap_lookupdev(errbuf);
        if (dev == NULL) {
                fprintf(stderr, "Couldn't find default device: %s\n", errbuf);
                return(2);
        }
        printf("Device: %s\n", dev);
        return(0);
}
```

```
root@raspberrypi:/home/pi/session07# gcc pcap_test01.c /usr/local/lib/libpcap.a
```